Synopsys Product Family:

SystemC Modeling Library Manual

Version 2.4.3
December 2017

Copyright Notice and Proprietary Information Notice

Destination Control Statement

Disclaimer

Trademarks

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043

www.synopsys.com

Contents

Preface

The preface of the *SystemC Modeling Library Manual* describes:

- About This Manual
- Documentation Conventions
- Terminology
- References

About This Manual

This manual describes SystemC Modeling Library 1 (SCML1) and SystemC Modeling Library 2 (SCML2) modeling objects. It also provides the guidelines on how to use SCML to create virtual prototype models. SCML stands for *SystemC Modeling Library*; it is a C++ library of modeling components built on top of TLM2.0 and SystemC, which are modeling libraries as well.

Note

- This version of the source code SCML kit is aligned with the SCML delivered in the Product Version N-2017.12 of Platform Architect MCO and Virtualizer.

It is assumed that you have some knowledge of SystemC.

This manual is organized as follows:

- Introduction gives an overview of the modeling objects and describes header files to be included.
- Memory Objects describes the SystemC Modeling Library (SCML) modeling objects.
- Clock Objects describes the SystemC Modeling Library (SCML) clock objects.
- Modeling Utilities describes the SCML modeling utilities.
- Functional Coverage describes the SCML functional coverage reference for SystemC TLM modeling.
- Modeling Guidelines explains the guidelines for LT modeling.

Documentation Conventions

This section lists and explains the documentation conventions used throughout this manual.

Convention	Description and Examples
italic	Is used in running text for: • GUI elements. For example: The *Enumeration* field contains a space-separated list of values. • New terms. For example: A *protocol library* is a collection of protocol definitions. • Web sites. For example: For more information, see *www.eclipse.org*. • E-mail addresses. For example: Please contact customer support via e-mail at *vp_support@synopsys.com*. • Manual names. For example: The preface of the *Analysis Manual* describes:

`courier`	Is used for:
	• Code text. For example:
	`list_library_configurations`
	`myConfig`
	In this example, `myConfig` is used.
	• System messages. For example:
	`JVM not found.`
	• Text you must type literally. For example:
	At the prompt, type `go`.
	• Names (of environment variables, commands, utilities, prompts, paths, macros, and so on). For example:
	The `build-options` command sets build parameters.

`courier italic`	Indicates variables. For example: *`scope`* specifies a module, a channel, or a refined port.
bold	Serves to draw your attention to the text in question. For example: `coreId = cwrSAGetCoreId("`**`mycore`**`");`

`[ ]`	Square brackets enclose optional items. For example:
	`clean [-pch]`
	If you must type a square bracket as part of the syntax, it is enclosed in single quotes. For example:
	`'['--use-vector']'`

Convention	Description and Examples

`{ }`	Braces enclose a list from which you must choose one or more items. For example:
	`add {signalPattern | portPattern} ID`
	If you must type a brace as part of the syntax, it is enclosed in single quotes. For example:
	`DECLARE '{'`
	`Item1`
	`Item1`
	`'}'`

`|`	A vertical bar separates items in a list of choices. For example: `autoflush {on | off}`
>	A right angle bracket separates menu commands. For example: The *Project > Update System Library* menu command is available.

`...`	A horizontal ellipsis in syntax indicates that the preceding expression may have zero, one, or more occurrences. For example:
	`build-options -option optionArgs ...`
	A horizontal ellipsis in examples and system messages indicates material that has been omitted. For example:
	`::scsh> dtrace add top1.signal_* $t1`
	`::scsh> dtrace add top1.clk_* $t1`
	`...`
	`::scsh> dtrace flush *`

Terminology

AT	In the context of PV, AT stands for Address Type. In the context of TLM2, AT stands for Approximately Timed.
API	Application Programmer's Interface
AV	Architect's View
DMA	Direct Memory Access
DT	Data Type
IP	Intellectual Property
LT	Loosely Timed
ASI TLM WG	Accellera Systems Initiative Transaction-Level Modeling Work Group
PODT	Plain Old Data Type
PV	Programmer's View
SOC	Stands for System-On-a-Chip.
SCML1	SystemC Modeling Library 1
SCML2	SystemC Modeling Library 2
STL	Socket Transaction Language
TLM	Transaction-Level Modeling
VPU	Virtual Processing Unit

References

This manual focuses on the use of SystemC, TLM2.0, and SCML for the creation of virtual prototype models. For more details on other use cases and detailed semantics of the libraries, see the following manuals:

- *SystemC Language Reference Manual,* IEEE standard 1666
- *IEEE Std. 1666 TLM-2.0 Language Reference Manual*

For details on interconnect components and the integration of processor models, see the *Integrating Third-Party Instruction-Accurate Models* manual.

Chapter 1 Introduction

This chapter describes:

- SCML2 Introduction
- Thread Safety
- Header Files
- SCML2 Modeling Objects

1.1 SCML2 Introduction

1.2 Thread Safety

The SCML2 API is not thread safe. This is important to take into account for debug API calls, since these are typically called from a different thread than the SystemC thread, and their implementation often uses data-structures shared with the non-debug API. In the OSCI environment, it is the caller's responsibility to ensure thread safety.

SCML2 is an easy-to-use abstraction layer on top of SystemC and TLM2. It hides a lot of the complexity and common code that is required to correctly manage TLM2 transactions and it provides with modeling objects that handle common aspects of Virtual Prototype modeling.

1.3 Header Files

All SCML2 header files can be included by including `scml2.h`. The SCML2 headers enable the Memory objects as well as the Model interface APIs and objects.

All SCML2 logging header files can be included by including `scml2_diagnostics.h`.

The utility objects as well as the deprecated modeling objects are available after the `scml.h` file has been included.

```
#include "scml.h"
```

The clock modeling objects are available after the `scml_clock.h` file has been included.

```
#include "scml_clock.h"
```

1.4 SCML2 Modeling Objects

SCML2 contains a wide range of modeling objects addressing different aspects of modeling. They are discussed in more detail in the coming sections.

1.4.1 Storage Modeling Objects

The storage modeling objects in SCML2 have been created in order to abstract and hide some of the tedious details of the TLM2.0 APIs while providing a simple mechanism to describe the memory map of a peripheral and to develop the behaviors that are associated with the different registers and bitfields of the component.

The following table provides an overview of the SCML2 modeling objects.

Table 1-1 SCML2 Modeling Objects

Modeling Object	Description
`memory`	Models memories and register files.
`memory_alias`	Models an alias for a memory region of another `memory` or `memory_alias` object.
`reg`	A register; it models a `memory_alias` object of size `1`.
`bitfield`	Models an alias for a number of consecutive bits in a `reg` object.
`router`	Models a dynamic address decoder that can map a memory region to a region in another `memory`, `router`, or `tlm2_gp_initiator_adapter` object.

All the modeling objects and global functions are part of the `scml2` namespace.

The following table provides an overview of the SCML helper functions.

Table 1-2 SCML2 Helper Functions

Helper Functions	Description
`memory_index_reference`	Is an intermediate object used when dealing with indices on `memory` and `memory_alias` objects.
`mappable_if`	Is the definition of the interface to the memory objects. It is used by the target adapter and the router object.

These modeling objects are structured hierarchically as follows:

- Memories:
  - `Memory`: Is the top-level entry for a component memory map. It is the only modeling object that maintains actual storage. This modeling object can have associated behavior.
  - `Memory_alias`: Refers to a section of the memory and can be used to define specialized behaviors per region or to provide with logical names for different sections in the memory map.
  - `Reg`: Is an alias of size one word.
  - `Bitfield`: Is a subword region.
- Router: is used to model programmable forwarding of memory accesses to different internal memories or initiator ports.

The memory storage objects have a default behavior that corresponds to the TLM2.0 blocking transport and debug APIs, plus they implement the `get_direct_memory_ptr` interface. Memories can be connected to the sockets via an adapter which implements a couple of conversions to make sure the SCML2 memory semantics can remain simple and also takes care of the non-blocking to blocking transport conversion if needed. Through this combination of features, it becomes extremely simple to build a memory model. Simply instantiating an SCML2 memory and connecting it to a TLM2.0 target socket via an adapter is sufficient. All TLM2.0 APIs are taken care of automatically and are of no worry to the developer. Adding aliases and registers allows the creation of a meaningful memory map for the component. All address decoding will be automatically taken care of. Finally by registering callbacks to implement the register behavior it is very easy to construct the functional model of a component.

There are a set of default behaviors available that can be registered with the memory objects. These are behaviors like: `ignore`, `read_only`, `write_only`, `clear_on_read`, `word_only`, `error`, `set_on_read`, `clear_on_write_1`, `clear_on_write_0`, `set_on_write_1`, `set_on_write_0`. Some of these are only available for the register objects. For a complete list and API details, see "Memory Objects" on page 17.

As already mentioned, it is possible to override the default behavior (which is storage) by registering a callback with a storage object. The following callback types are available:

- `Transport callback`: This is a callback that reuses the TLM2.0 transport API arguments and gives full control over the interpretation and handling of the transaction payload. This can be used when additional extensions of the payload need to be accessed which are not handled by the storage objects.
- `Read and Write callbacks`: are callbacks that only override the read or write behavior. Various versions of these callbacks exist: they have a data pointer as argument, but variants with and without `byte_enables`, `sc_time` argument exist. There is also a tagged version that can be used to pass the index of the memory element that is accessed into the callback. The return value of the callbacks is the `tlm_response_status` attribute as used in the payload. The storage object will pass this back via the transaction payload.
- `Debug`: An API to override the default debug behavior of that storage object.

Callbacks can be registered and unregistered so that context-specific behavior can be enabled, or also as speed optimization when there is only need to have a certain behavior when the component is in a specific state. Callbacks disable DMI behavior and as a consequence have an impact on the simulation speed. This is typically not an issue since callbacks are typically associated with peripheral components that are accessed infrequently (compared to memory and caches).

It is possible to create additional "custom" callbacks that can be reused for different modules by creating a class that derives from `scml2::memory_callback_base` and that implements the `execute(payload, sc_time)` interface.

Callbacks are related to the LT coding style, this means it is allowed to call `wait()` in the callback implementation and there are different synchronization possibilities that can be indicated when a callback is registered.

- `NEVER_SYNCING`: This means that the call is non blocking by nature and that it will never call wait.
- `SELF_SYNCING`: This means the callback is blocking and might call wait.
- `AUTO_SYNCING`: In this case, the callback is blocking and may call wait. The memory object will synchronize with the SystemC time before calling the callback. The timing annotation passed to the callback will always be `SC_ZERO_TIME`.

For a more detailed overview of every callback and access function that is available for the storage objects, see "Memory Objects" on page 17.

1.4.2 Timing and Synchronization

The timing and synchronization modeling objects are mostly related to the clocked modeling style and provide ease of use features to align the execution of models with clock boundaries since timing annotation on the TLM2.0 interfaces is not guaranteed to be aligned with the internal clock of a model.

The SCML modeling style defines a set of additional clock objects and interfaces, but is created such that compatibility with the traditional SystemC clocked modeling style is maintained. Traditionally in SystemC, a clock connection is represented by a boolean signal. This means that clock ports will be represented by a `sc_in<bool>` port in SystemC. The SCML modeling style maintains this style. So all clock connections are preferably done through signals and `sc_in/out<bool>` ports.

For full compatibility, clock generators should provide with an `scml_clock` object (see below) to provide the signal interface on the ports. This coding pattern allows to mix and match components that use a traditional SystemC clocked modeling style and the SCML clock modeling style.

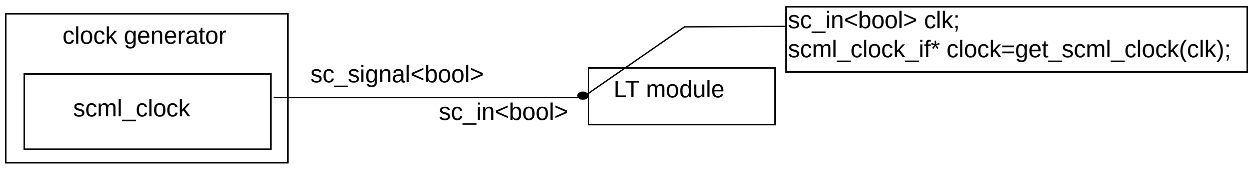
To retrieve the incoming clock from an `sc_in<bool>` input, the following API is provided:

```
// sc_in<bool> -> SCML clock
scml_clock_if* scml2::get_scml_clock(sc_core::sc_in<bool>& clk,
                                     bool allow_stubbed=false);
```

The API will retrieve the corresponding `scml_clock_if*` for the incoming clock. The second argument `allow_stubbed` controls the behavior if the signal interface is not bound to an SCML clock. If it is `false` or not provided and no SCML clock could be retrieved from the signal interface, then the API will fail by printing an error message and aborting the simulation. If `allow_stub` is set to `true`, it is possible to stub the input signal by binding it to any `bool` output port. In that case, the API will return a `NULL` pointer when no clock could be retrieved from the signal interface.

This coding pattern is supported by the `GenericIPLib` clock generator that is provided with Platform Creator. It is very much advised that each component that has a clock output follows the same pattern to ensure that there is a signal interface as well as an `scml_clock` object available for each clock connection.

Figure 1-1 The Coding Pattern



The following clocked modeling objects are provided for the SCML modeling style:

Table 1-3 Clocked Modeling Objects

Modeling Object	Description
`scml_clock`	Implements `sc_clock_if`. It is an optimized version of `sc_clock`.
`scml_clock_gate`	This is a module which takes a clock and an enable as inputs and produces a gated clock as output.
`scml_divided_clock`	This is a clock derived from another clock by multiplying the start time and/or the period with specified integer factors.
`clocked_module`	This is the base class for modules that want to receive SCML clock tick callbacks.
`clocked_callback`	This is a convenience class that forwards a clock tick callback to any member function of a module without the need to inherit from the `clocked_module` base class.
`clocked_timer`	This is a modeling object that provides a timer callback mechanism based on an SCML clock.
`clocked_event`	This is a convenience class that allows a SystemC method or thread to wait until a certain clock tick happens.

Table 1-3 Clocked Modeling Objects

Modeling Object	Description
`clocked_peq_container`	This is a modeling object for TLM2 models using the non-blocking APIs. It buffers payload arriving in the model, like multiple outstanding transactions, possibly coming with different timing annotations from different initiators.
`clocked_peq`	This is a modeling object similar to the `clocked_peq_container` that can trigger a callback whenever an element from the payload buffer becomes available.

A detailed reference documentation of the clocked modeling objects can be found in "Clock Objects" on page 67.

1.4.3 Utility Objects

SCML2 also contains a set of ease of use objects that deal with various aspects of creating a component model. Since they do not really fit one of the main categories (`timing` and `synchronization`) an overview for all of them is given in this section.

The following table provides an overview of the SCML2 utility objects.

Table 1-4 SCML2 Utility Objects

Modeling Object	Description
`tlm2_gp_target_adapter`	Allows a `memory` object to bind to a `tlm2_target_socket`.
`tlm2_gp_initiator_adapter`	Allows a `router` object to map a memory region to a region on a `tlm2_initiator_socket`.
`dmi_handler`	Is a convenience object to do the bookkeeping of DMI pointers.
`initiator_socket`	Is a convenience socket that first tries to do a DMI access before doing a bus access.
`status`	Is a simple object that holds a status value in string format.
`stream`	Is the front-end object of SCML2 logging library. It formats the output and sends it to the back-end logger objects for processing.
`severity`	Holds a severity name and a value. Lower severity values mean a higher severity level.

For more details on the objects, see "TLM2 Adapters" on page 63 and "Modeling Utilities" on page 95.

Chapter 2
Memory Objects

This chapter describes:

2.1 Introduction

The SCML memory objects serve several purposes:

- They provide with a default implementation of the TLM2 interfaces and generic payload.
- They provide with a mechanism to describe the memory map of a component.
- They should be used to model simple storage objects in a component.
- They have the necessary hooks to model the behaviors that are associated with an access to the elements in the memory map of a component.

The SCML memory objects only implement the Loosely Timed API's of the TLM2 interface, so they cannot interface directly with the TLM2 sockets, a `port_adapter` is required to handle timing and protocol conversion features of FT models (see "Port Adaptors" on page 77).

The interface used by the SCML memory objects is the `mappable_if` (see "mappable_if" on page 57).

2.2 Overview

This section provides a short overview of the different memory objects and their common features.

2.2.1 Transaction Routing from Socket to Memories

The `scml2::memory` is the top-level object to model the internal storage of a component, the other objects specify sub-regions within a memory. An `scml2::memory` can be bound to a target socket via a `port_adapter`. Multiple target sockets can be bound, each with its own `port_adapter`, to the same memory. All target sockets will see the memory at the same address.

To model multiple memory regions in a component, or when storage is seen at different locations from different sockets, an `scml2::router` can be used.

The `scml2::router` object is used to implement local address decoding and/or transaction forwarding in a model. A `router` can be bound to a target socket via a `port_adapter`. The `router` will maintain a map of address regions for the target port. Each address region is associated with a memory or initiator port to which the transactions in that address region need to be forwarded. This allows to connect multiple memories to a single target port at different addresses, but it also allows complex and dynamic transaction routing to memories and initiator ports.

2.2.2 Memory Map Modeling

The different memory objects can be used to model the memory map of a component.

The starting point for a memory map is the `scml2::memory` object. This object provides with the basic storage implementation, the handling of the TLM2 API's and the semantics of the generic payload. It is not possible to create a memory map consisting only of registers or bitfields, these objects need the `scml2::memory` object to convert generic payload transactions into simple register and bitfield accesses.

All other memory objects need an `scml2::memory` as parent object (directly or indirectly). A `memory_alias` represents a subrange in a memory and takes either a memory or another alias as parent. The address range represented by the alias should be smaller or have the same size as its parent object. The `scml2::reg` is intended to represent a single word in a memory map and can have a memory or an alias as parent. A bitfield represents a range of bits within a register, it can only have a register as parent.

Each memory object is templatized by its datatype (`<DT>`), the datatype defines the size of the individual elements (or words) in the memory object. The size of that datatype (in bytes) is also used to determine the address range represented by the memory object. The `scml2::memory` object determines the base size for all objects in its hierarchy. It is possible, for example, registers in a memory to have a word size that is a multiple of the word size of the memory. This allows to model memory maps with varying sized registers, in such a case it is easiest to work with an `scml2::memory` with datatype `unsigned char` which allows registers of any other supported datatype to be used in that memory map.

2.2.3 Properties and Attributes

All memory objects have a name and a width. The width gives the size of the datatype in `bytes`. They also have a DMI attribute to control whether the TLM2 direct memory interface is enabled for that object. Each object can control this individually, a hierarchical object will forward its setting to all its children.

An `scml2::memory` has latency parameters. These are latency parameters that apply for all objects in that memory hierarchy.

All objects, except `scml2::reg` have a size attribute and all child object have a reference to their parent object.

2.2.4 Behavior

To access the content of the storage objects there are multiple APIs available to support the different situations where memory access could be required. The variants differ in the way callbacks are triggered and debugger watchpoints are intended to be triggered.

Table 2-1 Behavior Type and Available APIs

Type	API	Callback	Watchpoints
Simple	`Put/get`	No	Yes
Simple-Debug	`Put/get_debug`	No	No

Table 2-1 Behavior Type and Available APIs

Type	API	Callback	Watchpoints
Trigger-Callbacks	`Put/get_with_triggering_callbacks`	Regular	Yes
Trigger-Debug-Callbacks	`Put/get_with_triggering_debug_callbacks`	Debug	No

For each variant, there are also different signatures available.

Table 2-2 Different Signatures of Each Variant

Style	Arguments
TLM2	`address`, `dataPtr`, `data_length`, `byte_enablePtr`, `enableLength`
TLM2-Word	`address`, `dataPtr`, `data_length`
Word	`index`, `DT`
Sub-Word	`index`, `DT`, `size`, `offset`

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

Table 2-3 Access Methods Supported by Memory Objects

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

2.2.5 Callbacks

The SCML memory objects provide with callbacks to model the behaviors that are associated with an access to the elements in the memory map of a component. Callbacks are methods in the component class that are registered with a memory object and that will be executed whenever there is an incoming transaction to the address range of that memory object. Callbacks can also be triggered via a Trigger-Callbacks or Trigger-Debug-Callbacks type of access to the memory object (see "Behavior" on page 18).

2.2.5.1 Callback Properties

In the memory map of a model, there can be multiple memory objects that cover the address range accessed by a transaction (for example, the top-level memory as well as a register). In such a case, SystemC Modeling Library will do the following:

1. To execute all behaviors that should be triggered by the transaction, the transaction will be "unrolled" to fit with the address range of the memory object with a callback: a burst access will be unrolled into accessed to the individual registers in the range, and an access to a register will be split into accesses to the bitfields it contains (provided at least one of them has a callback).

2. Only the callback of the "most refined" memory object will be called. That is the callback for the memory object that sits deepest in the object hierarchy (and that has a callback registered). This rule does not apply to bitfields, this means that when a register as well as one of its bitfields have a callback, then the register callback will be executed. It is up to the user to trigger the execution of the bitfield callbacks from the register callback (see "Operators" on page 42).

Following are the additional properties of callbacks:

- Callbacks can be registered either for debug or regular accesses and can be specialized for read or write accesses
- The functions implementing a callback will be called as part of the TLM2 LT transport interface implementation. This means that calling `wait()` is allowed in a regular callback.
- In a callback, the address passed in the payload argument will be adjusted so that it is relative to the start of the memory object to which the callback is registered.
- Registering a callback (regular callback or a debug callback) to a memory object will disable DMI access to this memory object.

It is possible to extend SCML with additional callback mechanisms (see "Callback Base Classes" on page 58).

2.2.5.2 Registering Callbacks

Several convenience functions are defined to register predefined callbacks to a memory object. These functions are defined in the `scml2/memory_callback_functions.h` and `scml2/memory_debug_callback_functions.h` files, respectively.

The following functions are available to register a member method as a callback to a memory object:

Regular callback registration

```
set_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_read_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_write_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_word_read_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_word_write_callback(mem, SCML2_CALLBACK(method), syncType, tag)
set_post_write_callback(mem, SCML2_CALLBACK(method), syncType, tag);
```

The following functions are available to register a member method as a debug callback to a memory object:

Debug callback registration

```
set_debug_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_read_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_write_callback(mem, SCML2_CALLBACK(method), tag)
```

Where the arguments represent the following:

mem	Is the memory object to which the callback will be registered.
`SCML2_CALLBACK(`*method*`)`	Is macro that helps pass the callback into the registration function. method is the name of the callback method, which should be a member of the model class of the memory object. The supported signatures of the callback method are explained in the sections of the relevant memory objects.

syncType	Can be one of the following: • NEVER_SYNCING indicates that the callback is nonblocking and must never call wait(). • SELF_SYNCING indicates that the callback is blocking and may call wait(). The timing annotation is passed unmodified to the callback. • AUTO_SYNCING indicates that the callback is blocking and may call wait(). The memory object synchronizes before and after calling the callback. The timing annotation passed to the callback is always SC_ZERO_TIME. These types are defined in the scml2/types.h file. The Post predefined behavior callbacks do not support SELF_SYNCING callbacks (since the callback does not have a time argument)
tag	Is an optional argument. When provided it is a user-provided integer that is passed to the callback. For example, so that the same callback can be attached to all registers in a register array.

2.2.5.3 Removing Callbacks

Callbacks can be removed at runtime. This is usually not needed, but to improve simulation performance it is a good idea to disable the callbacks for memory objects that are very often accessed (for example, in a polling loop) but where the behavior that is modeled in the callback is rarely needed. In such a case, a performance improvement is to remove the callback and register it again when the external event that the polling loop is checking for has happened.

The following API's are available to unregister callbacks from a memory object:

Unregistering regular callbacks

```
void remove_callback()
void remove_read_callback()
void remove_write_callback()
```

Unregistering debug callbacks

```
void remove_debug_callback()
void remove_debug_read_callback()
void remove_debug_write_callback()
```

2.2.5.4 Callback Methods

Every callback registration function accepts a few signatures for the callback methods. For a detailed list of the callback signatures, see sections: “Callbacks” on page 31, “Callbacks” on page 37, “Callbacks” on page 43, “Callbacks” on page 49, and “Callbacks” on page 55.

2.2.6 Access Restrictions

Access restrictions are a special type of callbacks. They are intended to control whether transactions can access memory objects or parts of them. Typically, access to certain registers or bitfields in a memory map might be controlled by other registers or bitfields to prevent that the component is pushed into an illegal state through some external input.

Such behavior could be modeled using regular callbacks, but it can be tedious due to the "most refined" rule that applies to regular callbacks. Often the restriction applies for all bitfields in a register, forcing you to add the access check to all bitfield callbacks, or to replace the bitfield callbacks with a register callback.

2.2.6.1 Properties of Access Restrictions

Access restrictions have the following properties:

1. Access restriction callbacks are executed for an incoming transaction before any of the behavior (regular) callbacks are executed.
2. Access restrictions are executed for all memory objects in the memory map that have an address range that overlaps with the address range of the transaction.
3. To execute all registered access restrictions, the transaction will be "unrolled" to fit with the address range of the memory object with an access restriction: a burst access will be unrolled into accessed to the individual registers in the range, and an access to a register will be split into accesses to the bitfields it contains (provided at least one of them has an access restriction).
4. An access restriction determines which part of an access is restricted allowed by manipulating the `byte_enable` argument in the TLM2 Generic Payload. The byte enable uses `0xFF` per byte, for access restrictions the limitation that only value `0x0` and `0xFF` are allowed is lifted and the `byte_enable` value is used as a bit enable.
5. Access restrictions are cumulative: All access restrictions on the path to a certain memory object are executed. Each of them can further modify the `byte_enable` argument of the transaction.
6. When access is disabled for all bits of a memory object, the corresponding behavior callback will not be executed (this includes the default behavior of a memory object).
7. The restrict callback should take care of the data in the payload and the storage to ensure that any callback that might be executed works with the expected value. For example, in case a bitfield has an access restriction, but there is also a callback on the parent register, the access restriction may want to modify the transaction data so that the register callback gets the current value of the bitfield rather than what is specified in the original transaction payload.
8. The return value of an access restriction is used to update the transaction response field. The return type of a restrict callback is `scml2::access_restriction_result`. Possible values are:

`scml2::RESTRICT_NO_ERROR`	No change in transaction response, it is up to the callbacks to finalize the value (both restrict callbacks as well as behavior callbacks).
`scml2::RESTRICT_ERROR`	The transaction response is modified to `TLM_GENERIC_ERROR_RESPONSE` and the transaction is immediately aborted. No further restrictions are checked and no behavior callbacks are executed.

2.2.6.2 Registering a Restriction

Several convenience functions are defined to register access restrictions to a memory object. These functions are defined in the `scml2/memory_restriction_functions_include.h` and `scml2/bitfield_restriction_functions_include.h` file, respectively.

The following functions are available to register a member method as an access restriction for a memory object:

Access restriction registration

```
set_restriction(mem, SCML2_CALLBACK(method), tag)
set_read_restriction(mem, SCML2_CALLBACK(method), tag)
set_write_restriction(mem, SCML2_CALLBACK(method), tag)
```

Where the arguments represent the following:

`mem`	Is the memory object to which the callback will be registered.
`SCML2_CALLBACK(`*`method`*`)`	Is the macro that helps pass the callback into the registration function. `method` is the name of the callback method, which should be a member of the model class of the memory object. The supported signatures of the callback method are explained in section "Access Restriction Methods" on page 23.
`tag`	Is an optional argument. When provided, it is a user-provided integer that is passed to the callback. For example, so that the same callback can be attached to all registers in a register array.

This is similar to the registration of a regular behavior callback with the exception that restrictions do not have a *`syncType`* as they are not allowed to influence the timing of the transaction.

2.2.6.3 Removing an Access Restriction

Access restrictions can be removed with the following API's:

API's to remove access restrictions

```
scml2::remove_ restriction();
scml2::remove_read_restriction();
scml2:: remove_write_restriction();
```

2.2.6.4 Access Restriction Methods

There are two forms of access restriction methods supported:

- A simple one available for registers, and bitfields, and
- a TLM2-based restriction method available for all memory objects.

The signature for these variants is as follows:

```
scml2::access_restriction_result MyRestrictFtion(DT& data, DT& bit_enables, int tag)
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)
```

With the following arguments:

Table 2-4 Access Restriction Methods - Arguments

Argument	Description
`DT& data`	Specifies the data argument, which is used to represent the data from the payload from the parts of the access that are not restricted, that is, the parts of the transaction that should be processed in the behavior callbacks. The data that should be assumed by the callbacks for the parts of the access that are restricted. This is the data that is already processed by the restriction.
`DT& bit_enables`	Indicates which parts of the data vector are restricted (corresponding bits set to zero). This will influence whether callbacks are executed. Callbacks are not executed if all `bit_enables` are zero for the object the callback is associated with.
`Int `*`tag`*	Is an optional argument. A tag can be set when registering a callback, that same tag value will be passed to the access restriction each time it is called.

Table 2-4 Access Restriction Methods - Arguments

Argument	Description
`tlm::tlm_generic_payload& trans`	Is the TLM transaction payload with updated `byte_enables` and address value (address is always relative to the base-address of the memory object), also `streamin_width` is already handled and converted into `byte_enables`. This variant gives access to all other payload attributes to decide on the restriction. It can be used to check for extensions or to deal with burst accesses.
`scml2::access_restriction_result`	The return value should be either `scml2::RESTRICT_NO_ERROR` or `scml2::RESTRICT_ERROR`. In the latter case, the transaction is immediately aborted and a `TLM_GENERIC_ERROR_RESPONSE` will be returned.

2.2.6.5 Predefined Access Restrictions

For a number of frequent access restriction types, a predefined SCML2 API is provided to set a restriction on a memory object:

Predefined Access Restrictions

```
scml2::set_ignore_restriction(mem, DT value = 0);
scml2::set_error_restriction(mem);
scml2::set_word_restriction(mem);
scml2::set_read_ignore_restriction(mem, DT value = 0);
scml2::set_read_error_restriction(mem);
scml2::set_read_word_restriction(mem);
scml2::set_write_ignore_restriction(mem, DT value = 0);
scml2::set_write_error_restriction(mem);
scml2::set_write_word_restriction(mem);
```

These API's have the following properties:

- The predefined access restrictions can be set for all accesses, or only for read or write accesses.
- The predefined access restrictions can be set on any memory object (except the word-access limitation that cannot be set on a bitfield).
- The error restriction will set the transaction response to `TLM_GENERIC_ERROR_RESPONSE`.
- The value argument for the `scml2::set_read_ignore_restriction` will be stored in the data pointer of the transaction payload.
- The value argument for the `scml2::set_write_ignore_restriction` should be set to the value of the memory object for which the restriction is set, in order to make sure that any callback on a memory object higher up in the memory map hierarchy does not overwrite that memory object.
  - So typically, `scml2::set_write_ignore_restriction(mem, (DT)mem);`
  - Other values are possible, for example, passing `0` as value will implement a `set_0_on_write` while still allowing for other behavior callbacks to be executed.
- A word access restriction will allow any transaction that is,
  - Word aligned
  - Data-length is a multiple of word-size
  - All `byte_enables` are set

This means that an aligned 128-bit access in a 64-bit memory object will be allowed. Multiple register callbacks could be triggered, but all can assume the access is for the full word.

2.2.6.6 Access Restrictions Implementation Helper Functions

To facilitate the implementation of an access restriction, a number of helper functions are available that represent typical functionality in a restrict callback:

Access restriction helper functions:

```
scml2::restrict_all(DT& data, DT& bit_enables, DT& restrict_value = 0);
scml2::restrict_all<DT>(tlm::tlm_generic_payload& trans,
        DT restrict_value = 0);
scml2::restrict_all_and_store(DT& data, DT& bit_enables, DT& restrict_value,
        MEM_OBJECT<DT>& mem_obj);
scml2::restrict_some(DT& data, DT& bit_enables, DT& restrict_value,
        DT in_data_mask);
scml2::restrict_some_and_store(DT& data, DT& bit_enables,
        DT& restrict_value, DT in_data_mask,
        MEM_OBJECT<DT>& mem_obj);
```

Properties of the helper functions:

- `restrict_all` will set `bit_enables` to `0` and pass the `restrict_value` as data (by default set to `0`).
- `restrict_all_and_store` will set `bit_enables` to `0`, pass the `restrict_value` as data and store the restrict value in the memory object.
- `restrict_some` can be used to partially restrict the access. The `data_mask` should indicate the valid bits in the data (bits set to `1` in the `data_mask` will be restricted, that is, `bit_enables` will be set to `0`). The `restrict_value` will be passed as data.
- `restrict_some_and_store` can be used to partially restrict the access. The `data_mask` should indicate the valid bits in the data (bits set to `1` in the `data_mask` will be restricted, that is, `bit_enables` will be set to `0`). The `restrict_value` will be passed as data and also will be stored in the memory object.
- All convenience functions return `scml2::RESTRICT_NO_ERROR`.

2.2.6.7 Access Restrictions Versus Behavior Callbacks

- Restrictions
  - All restrictions along the path of a transaction are checked: All elements in the memory hierarchy to the leaf nodes that get accessed can add restrictions.
  - Restrictions should manipulate `bit_enables/byte_enables` to implement the restrictions.
  - Are only allowed to further restrict (not to overwrite restrictions elsewhere in the path).
  - Restrictions should manipulate payload data to ensure consistency independent of other restrictions and behavior callbacks.
- Behavior callbacks
  - Only the 'most specified' behavior gets executed.
  - Behaviors have full access/control over transaction payload.
  - No need to care about any other callbacks or restrictions that might get executed.

- Behavior callbacks will not be executed when `bit_enables`/`byte_enables` are all disabled for the memory object.
- 'Word'-style behavior callbacks will not check whether all `byte-enables` are set if there is a restriction somewhere on the path to the callback.

2.2.7 Vectors of Memory Objects

When managing vectors of memory objects, one can use the `scml2::vector` class which is API-compatible with `sc_core::sc_vector`, but has a different naming convention for the objects created in the array.

The names will be '<*name*>_0, <*name*>_1, …, <*name*>_n'. For `sc_core::sc_vector`, these names would be '<*name*>, <*name*>_1, …, <*name*>_n'.

Except for this difference, the two classes are intended to have indentical behavior.

You can also use the `scml2::vector` class for other modeling objects (such as ports) to instantiate vectors of objects with the adapted naming convention.

2.3 memory

The `scml2::memory` represents the top-level object for the local memory or register file description of a model. The memory does the actual storage allocation and can be bound to a TLM2 target socket via an adapter. It implements the `mappable_if` interface which requires and object to implement the TLM2 LT interfaces (`b_transport`, `transport_dbg` and `get_direct_mem_ptr`). The `mappable_if` is provided in SCML2 as a link between the storage objects and the TLM2 socket interfaces. For details, see “mappable_if” on page 57. A memory can have aliases and/or registers to specify the detailed memory hierarchy of the component.

The include file of the memory objects is `scml2/memory.h`.

It has the following properties:

- `name` and `size`, as provided with the constructor of the object. The name of the memory will be used by the debug and analysis tools to refer to this object.
- `width`: representing the datawidth that is stored in the memory. This is derived from the datatype template argument for the class.
- The `scml2::memory` also has a default read and write latency which will be used in the implementation of the default behavior. Any `b_transport` read or write call will get the latency implemented accordingly through timing annotation. This value will also be passed when the storage pointer is made available on a DMI request from an initiator. It is the only object with this parameter.
- The memory object also implements the index operation (`[]`) and has an initialize call to define an initial value for the storage.

2.3.1 Types

The memory class is templated with the underlying value type:

```
Template <typename DT> class memory
```

The following types are supported:

Table 2-5 Supported Data types

Supported Data types	Word size
`unsigned char`	8
`unsigned short`	16
`unsigned int`	32
`unsigned long long`	64
`sc_dt::sc_biguint<128>`	128
`sc_dt::sc_biguint<256>`	256
`sc_dt::sc_biguint<512>`	512

2.3.2 Constructors

The following constructor is available

```
memory(const std::string& name, unsigned long long size);
```

This creates a new memory. The size argument must be specified in words.

2.3.3 Properties

The following methods are available to access the properties of a memory:

- `const std::string& get_name() const`

  Returns the full hierarchical name of the memory object.

- `unsigned long long get_size() const`

  Returns the size of the memory object in words.

- `unsigned int get_width() const`

  Returns the width in bytes of the underlying data type of the memory object.

-
  ```
  void set_default_read_latency(const sc_core::sc_time& t)
  const sc_core::sc_time& get_default_read_latency() const
  void set_default_write_latency(const sc_core::sc_time& t)
  const sc_core::sc_time& get_default_write_latency() const
  ```

  Sets/gets the latency returned in the `tlm::tlm_dmi` structure of the `get_direct_mem_ptr()` call. If no callback is attached, this latency is also added to the timing annotation argument of the `b_transport()` call.

- `bool is_dmi_enabled()`

  Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

-
  ```
  void enable_dmi()
  void disable_dmi()
  ```

  Enables/disables DMI accesses for the object.

- `const std::string& get_description() const`
  `void set_description(const std::string&)`

  Gets/sets the description for the memory object. This description can be displayed in debuggers.

- `bool has_default_read_behavior() const;`
  `bool has_default_write_behavior() const;`
  `bool has_default_debug_read_behavior() const;`
  `bool has_default_debug_write_behavior() const;`
  `bool has_never_syncing_read_behavior() const;`
  `bool has_never_syncing_write_behavior() const;`
  `bool is_dmi_allowed() const;`
  `bool is_dmi_read_allowed() const;`
  `bool is_dmi_write_allowed() const;`

  These are a set of API's to query what type of behavior is associated with the memory and whether DMI is enabled.

- `bool has_default_restriction() const;`
  `bool has_default_read_restriction() const;`
  `bool has_default_write_restriction() const;`

  These API's query whether there is a restriction set for the memory.

The memory object can be bound to a TLM2 target socket via a `port_adapter`. For details, see Port Adaptors. A target port adaptor may be bound to an SCML2 memory, as shown below.

```
// bind adaptor to memory
scml2::memory my_memory("my_memory", 0x100);
(*my_port_adapter)(my_memory);
```

2.3.4 Behaviors

2.3.4.1 Initialization

The `initialize()` method can be used to put the specified initial value in the whole memory array:

```
void initialize (const DT& value = DT())
```

In case no argument is given, the value returned by the default constructor for the underlying data type is used.

2.3.4.2 Transport Calls

The following transport methods are available on the `memory` objects:

The `memory` object implements the following TLM2 methods:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans,
                                tlm::tlm_dmi& dmiData)
```

These methods trigger the callbacks registered to the memory object hierarchy represented by the memory.

The `memory` object also implements the following TLM2-like methods:

```
void transport_without_triggering_callbacks(tlm::tlm_generic_payload& trans)
void transport_debug_without_triggering_callbacks(
                                        tlm::tlm_generic_payload& trans)
```

These transport methods do not trigger any callbacks. They access the current content of the `memory` object. The debug methods also do not trigger any watchpoints on the `memory` object.

2.3.4.3 Put/Get Access

The `memory` object supports the different put/get behaviors as described in "Behavior" on page 18. The following set of access functions is supported according to the different types and styles as defined in "Behavior" on page 18:

Table 2-6 Pit/Get Access

Type	API	Callback	Watchpoints
`Simple`	`Put/get`	No	Yes
`Simple-Debug`	`Put/get_debug`	No	No
`Trigger-Callbacks`	`Put/get_with_triggering_callbacks`	Regular	Yes
`Trigger-Debug-Callbacks`	`Put/get_with_triggering_debug_callbacks`	Debug	No

For each variant, there are also different signatures available:

Table 2-7 Different Signatures of Each Variant

Style	Arguments
TLM2	`address`, `dataPtr`, `data_length`, `byte_enablePtr`, `enableLength`
TLM2-Word	`address`, `dataPtr`, `data_length`
Word	`index`, `DT`
Sub-Word	`index`, `DT`, `size`, `offset`

The following table describes the arguments mentioned in the table 2-7.

Table 2-8 Argument Descriptions

Argument name	C++	description
`address`	`unsigned long long address`	Byte address as in TLM2 GP.
`dataPtr`	`(const) unsigned char* data`	Data array as in TLM2 GP.
`data_length`	`unsigned int dataLength`	The length of the transaction in bytes as in TLM2 GP.
`byte_enablePtr`	`const unsigned char* byteEneblePtr`	Specifies a byte enable array which can be `0` as in TLM2 GP.
`enableLength`	`unsigned int byteEnableLength`	Length of the byte enable array in bytes as in TLM2 GP.

Table 2-8 Argument Descriptions

Argument name	C++	description
`index`	`unsigned long long index`	The word index as specified by the DT template of the memory object.
`DT`	`(const) DT& data`	Is the data.
`size`	`unsigned int size`	Is the size of the access in bytes.
`offset`	`unsigned int offset`	Is the offset for the access in bytes.

Additional notes:

- The Trigger- Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the read data instead of passing it as an argument.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be `0`) since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

Table 2-9 Memory Objects

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments, see the include file of the `memory: scml2/memory.h`.

2.3.4.4 Operators

The following assignment operators are available:

```
reference operator[](unsigned long long index)
DT operator[](unsigned long long index) const
```

The `lvalue` version of the *`index`* operator returns a `memory_index_reference` object that forwards all operations to the referenced memory object. The `const` version returns the current value.

```
iterator begin()
const_iterator begin() const
```

Returns a random access iterator pointing to the first element in the `memory` object.

```
iterator end()
const_iterator end() const
```

Returns a random access iterator pointing to the end of the `memory` object.

2.3.5 Callbacks

The memory object supports the callbacks listed below. For more information on callbacks, see also "Callbacks" on page 19.

2.3.5.1 Regular Callbacks

The memory object only supports the `set_callback` registration-style callback methods.

Regular callback registration

```
set_callback(mem, SCML2_CALLBACK(method), syncType, tag)
```

It accepts callback methods with the following signatures:

```
void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)
void transportCallback(tlm::tlm_generic_payload&, int tag)
```

Additional notes:

- In all the above callback methods, *`tag`* is an optional argument, only to be used when the callback is registered with a tag.
- The untimed callback (without the `sc_time` parameter) cannot be `SELF_SYNCING` callbacks.

2.3.5.2 Debug Callbacks

The memory object support the following registration API's for debug callbacks:

Debug callback registration

```
set_debug_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_read_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_write_callback(mem, SCML2_CALLBACK(method), tag)
```

For debug callbacks, method must have one of the following signatures:

```
unsigned int transportCallback(tlm::tlm_generic_payload&)
unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)
```

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, `0` must be returned.

2.3.6 Access Restrictions

The memory object supports access restrictions as described in "Access Restrictions" on page 21. The memory supports all predefined access restrictions as specified in"Predefined Access Restrictions" on page 24, but it only supports TLM2-style access restriction callbacks:

```
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)
```

2.4 memory_alias

The `scml2::memory_alias` object specifies a subregion of a `scml2::memory`. A `memory_alias` cannot exist on its own, it always needs a `scml2::memory` parent object. The alias does not come with its own storage and can have further aliases and/or registers. The alias cannot be bound to a TLM2 target socket (or the adapters) as it does not implement the `mappable_if`. The use of a `memory_alias` is to provide with a different name for a subregion to make sure the model reflects the design specification of the component, but also serves as a hook so that specialized behavior can be associated with part of the memory or register map of the component. The properties of an `memory_alias` are similar to the memory and provided via the `memory_base` base class.

The include file of the `memory_alias` objects is `scml2/memory_alias.h`.

It has the following properties:

- The `scml2::memory_alias` takes a name, parent memory, offset and size in the constructor. Offset indicates the start index for this alias within the range of the parent memory (or memory_alias).
- It has access functions for name, size, offset, parent and width. Exactly like these also exist for the `scml2::memory`.
- It also supports the index operator `[]` and the initialize call as exist for the memory.
- Callbacks that are associated with a `memory_alias` will override the behavior of the parent memory. Both the default behavior and the behavior that gets registered via the callback mechanism are overruled this way.

2.4.1 Types

The `memory_alias` class is templated with the underlying value type:

```
Template <typename DT> class memory_alias
```

The following types are supported:

Table 2-10 Supported Types

Supported Datatype	Word size
`unsigned char`	8
`unsigned short`	16
`unsigned int`	32
`unsigned long long`	64
`sc_dt::sc_biguint<128>`	128
`sc_dt::sc_biguint<256>`	256
`sc_dt::sc_biguint<512>`	512

The memory alias should have a datatype where the word size is a multiple of that of its parent memory.

2.4.2 Constructors

The following constructors are available:

```
memory_alias(const std::string& name,
             memory<DT>& parent,
             unsigned long long offset,
             unsigned long long size)
memory_alias(const std::string& name,
             memory_alias<DT>& parent,
             unsigned long long offset,
             unsigned long long size)
```

Create a new `memory_alias` object. The *`size`* and *`offset`* argument must be specified in words.

2.4.3 Properties

The following methods are available to access the properties of a memory_alias:

- `const std::string& get_name() const`

  Returns the full hierarchical name of the of the `memory_alias` object.

- `unsigned long long get_offset() const`

  Returns the offset in words relative to the top-level memory object.

- `unsigned long long get_size() const`

  Returns the size of the `memory_alias` object in words.

- `unsigned int get_width() const`

  Returns the width in bytes of the underlying data type of the `memory_alias` object.

- `memory_base* get_parent() const`

  Returns a pointer to the parent memory or `memory_alias` object.

- `bool is_dmi_enabled()`

  Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

-
  ```
  void enable_dmi()
  void disable_dmi()
  ```

  Enables/disables DMI accesses for the object.

-
  ```
  const std::string& get_description() const
  void set_description(const std::string&)
  ```

  Gets/sets the description for the `memory_alias` object. This description can be displayed in debuggers.

-
  ```
  bool has_default_read_behavior() const;
  bool has_default_write_behavior() const;
  bool has_default_debug_read_behavior() const;
  bool has_default_debug_write_behavior() const;
  bool has_never_syncing_read_behavior() const;
  bool has_never_syncing_write_behavior() const;
  bool is_dmi_allowed() const;
  bool is_dmi_read_allowed() const;
  bool is_dmi_write_allowed() const;
  ```

  These are a set of API's to query what type of behavior is associated with the `memory_alias` and whether DMI is enabled.

-
  ```
  bool has_default_restriction() const;
  bool has_default_read_restriction() const;
  bool has_default_write_restriction() const;
  ```

  These API's query whether there is a restriction set for the `memory_alias`.

2.4.4 Behaviors

2.4.4.1 Initialization

The `initialize()` method can be used to put the specified initial value in the whole memory array:

```
void initialize (const DT& value = DT())
```

In case no argument is given, the value returned by the default constructor for the underlying data type is used.

2.4.4.2 Transport Calls

The following transport methods are available on the memory objects:

The `memory_alias` object implements the following TLM2 methods:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) unsigned int
transport_dbg(tlm::tlm_generic_payload& trans)
```

These methods trigger the callbacks registered to the `memory_alias` object hierarchy represented by the memory.

The `memory_alias` object also implements the following TLM2-like methods:

```
void transport_without_triggering_callbacks(tlm::tlm_generic_payload& trans)
void transport_debug_without_triggering_callbacks(
                  tlm::tlm_generic_payload& trans)
```

These transport methods do not trigger any callbacks. They access the current content of the `memory_alias` object. The debug methods also do not trigger any watchpoints on the `memory_alias` object.

2.4.4.3 Put/Get Access

The `memory_alias` object supports the different put/get behaviors as described in "Behavior" on page 18. The following set of access functions is supported according to the different types and styles as defined in "Behavior" on page 18:

Table 2-11 Supported Access Functions

Type	API	Callback	Watchpoints
Simple	`Put/get`	No	Yes
Simple-Debug	`Put/get_debug`	No	No
Trigger-Callbacks	`Put/get_with_triggering_callbacks`	Regular	Yes
Trigger-Debug-Callbacks	`Put/get_with_triggering_debug_callbacks`	Debug	No

For each variant, there are also different signatures available:

Table 2-12 Different Signatures of Each Variant

Style	Arguments
TLM2	`address`, `dataPtr`, `data_length`, `byte_enablePtr`, `enableLength`
TLM2-Word	`address`, `dataPtr`, `data_length`

Table 2-12 Different Signatures of Each Variant

Style	Arguments
Word	`index`, `DT`
Sub-Word	`index`, `DT`, `size`, `offset`

The following table describes the arguments mentioned in the table 2-12.

Table 2-13 Argument Descriptions

Argument name	C++	Description
`address`	`unsigned long long address`	Byte address as in TLM2 GP.
`dataPtr`	`(const) unsigned char* data`	Data array as in TLM2 GP.
`data_length`	`unsigned int dataLength`	The length of the transaction in bytes as in TLM2 GP.
`byte_enablePtr`	`const unsigned char* byteEneblePtr`	Specifies a byte enable array which can be `0` as in TLM2 GP.
`enableLength`	`unsigned int byteEnableLength`	Length of the byte enable array in bytes as in TLM2 GP.
`index`	`unsigned long long index`	The word index as specified by the DT template of the memory object.
`DT`	`(const) DT& data`	Is the data.
`size`	`unsigned int size`	Is the size of the access in bytes.
`offset`	`unsigned int offset`	Is the offset for the access in bytes.

Additional notes:

- The Trigger-Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the read data instead of passing it as an argument.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be `0`) since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

Table 2-14 Supported Access Methods

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x

Table 2-14 Supported Access Methods

Style	TLM2	TLM2-Word	Word	Sub-Word
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments please check the include file of the `memory_alias`: `scml2/memory_alias.h`.

2.4.4.4 Operators

The following assignment operators are available:

```
reference operator[](unsigned long long index)
DT operator[](unsigned long long index) const
```

The `lvalue` version of the *`index`* operator returns a `memory_index_reference` object that forwards all operations to the referenced `memory` object. The `const` version returns the current value.

```
iterator begin()
const_iterator begin() const
```

Returns a random access iterator pointing to the first element in the `memory_alias` object.

```
iterator end()
const_iterator end() const
```

Returns a random access iterator pointing to the end of the `memory_alias` object.

2.4.5 Callbacks

The `memory_alias` object supports the callbacks listed below. For more information on callbacks, see also "Callbacks" on page 19.

2.4.5.1 Regular Callbacks

The `memory_alias` object only supports the `set_callback` registration-style callback methods.

Regular callback registration

```
set_callback(mem, SCML2_CALLBACK(method), syncType, tag)
```

It accepts callback methods with the following signatures:

```
void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)
void transportCallback(tlm::tlm_generic_payload&, int tag)
```

Additional notes:

- In all the above callback methods, `tag` is an optional argument, only to be used when the callback is registered with a tag.
- The untimed callback (without the `sc_time` parameter) cannot be `SELF_SYNCING` callbacks.

2.4.5.2 Debug Callbacks

The `memory_alias` object support the following registration API's for debug callbacks.

Debug callback registration

```
set_debug_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_read_callback(mem, SCML2_CALLBACK(method), tag)
set_debug_write_callback(mem, SCML2_CALLBACK(method), tag)
```

For debug callbacks, `method` must have one of the following signatures:

```
unsigned int transportCallback(tlm::tlm_generic_payload&)
unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)
```

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, `0` must be returned.

2.4.6 Access Restrictions

The `memory_alias` object supports access restrictions as described in "Access Restrictions" on page 21. The memory_alias supports all predefined access restrictions as specified in "Predefined Access Restrictions" on page 24, but it only supports TLM2-style access restriction callbacks:

```
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)
```

2.5 reg

The `scml2::reg` is a `memory_alias` of size `1`. That is, it represents a subrange of the memory which is the same size as the width of the memory or in other words represents one storage location of the same size as the datatype template argument of the storage classes. As with the `memory_alias`, it does not implement its own storage and must have a parent memory or memory alias and cannot be bound to the TLM2 target sockets (or adapters). On top of that it cannot have any other aliases and or registers, but it can have bitfields associated with itself. It is the leave node of the memory hierarchy. The register object has all the same properties and access methods as the `memory_alias` (name, parent, offset, index operator, callback overruling). The key additional feature for the register is that it can be used as a regular variable.

The include file of the reg objects is `scml2/reg.h`.

2.5.1 Types

The register class is templated with the underlying value type:

```
Template <typename DT> class reg
```

The following types are supported:

Table 2-15 Supported Datatypes

Supported Datatypes	Word Size
`unsigned char`	8
`unsigned short`	16
`unsigned int`	32
`unsigned long long`	64
`sc_dt::sc_biguint<128>`	128
`sc_dt::sc_biguint<256>`	256
`sc_dt::sc_biguint<512>`	512

The register should have a datatype where the word size is a multiple of that of its parent memory or `memory_alias`.

2.5.2 Constructors

The following constructors are available:

```
reg(const std::string& name,
    memory<DT>& parent,
    unsigned long long offset)
reg(const std::string& name,
    memory_alias<DT>& parent,
    unsigned long long offset)
```

Create a new `reg` object. The *`offset`* argument must be specified in words.

2.5.3 Properties

The following methods are available to access the properties of a register:

- `const std::string& get_name() const`

  Returns the full hierarchical name of the of the `reg` object.

- `unsigned long long get_offset() const`

  Returns the offset in words relative to the top-level `memory` object.

- `unsigned int get_width() const`

  Returns the width in bytes of the underlying data type of the `reg` object.

- `memory_base* get_parent() const`

  Returns a pointer to the parent `memory` or `memory_alias` object.

- `bool is_dmi_enabled()`

  Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

-
  ```
  void enable_dmi()
  void disable_dmi()
  ```

  Enables/disables DMI accesses for the object.

-
  ```
  const std::string& get_description() const
  void set_description(const std::string&)
  ```

  Gets/sets the description for the `register` object. This description can be displayed in debuggers.

-
  ```
  bool has_default_read_behavior() const;
  bool has_default_write_behavior() const;
  bool has_default_debug_read_behavior() const;
  bool has_default_debug_write_behavior() const;
  bool has_never_syncing_read_behavior() const;
  bool has_never_syncing_write_behavior() const;
  bool is_dmi_allowed() const;
  bool is_dmi_read_allowed() const;
  bool is_dmi_write_allowed() const;
  ```

  These are a set of API's to query what type of behavior is associated with the register and whether DMI is enabled.

-
  ```
  bool has_default_restriction() const;
  bool has_default_read_restriction() const;
  bool has_default_write_restriction() const;
  ```

These API's query whether there is a restriction set for the register.

2.5.4 Behaviors

2.5.4.1 Initialization

The `initialize()` method can be used to put the specified initial value in the whole memory array:

```
void initialize (const DT& value = DT())
```

In case no argument is given, the value returned by the default constructor for the underlying data type is used.

2.5.4.2 Transport Calls

The following transport methods are available on the memory objects:

The `reg` object implements the following TLM2 methods:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) unsigned int
transport_dbg(tlm::tlm_generic_payload& trans)
```

These methods trigger the callbacks registered to the `reg` object hierarchy represented by the memory.

The `reg` object also implements the following TLM2-like methods

```
void transport_without_triggering_callbacks(tlm::tlm_generic_payload& trans)
void transport_debug_without_triggering_callbacks(
                                           tlm::tlm_generic_payload& trans)
```

These transport methods do not trigger any callbacks. They access the current content of the `reg` object. The debug methods also do not trigger any watchpoints on the `reg` object.

2.5.4.3 Put/Get Access

The `reg` object supports the different put/get behaviors as described in section "Behavior" on page 18. Unlike the `memory` object, the `put` and `get` methods of the `reg` object do not take an index argument (since this should always be `0`). The following set of access functions is supported according to the different types and styles as defined in "Behavior" on page 18:

Table 2-16 Supported Access Functions

Type	API	Callback	Watchpoints
Simple	`Put/get`	No	Yes
Simple-Debug	`Put/get_debug`	No	No
Trigger-Callbacks	`Put/get_with_triggering_callbacks`	Regular	Yes
Trigger-Debug-Callbacks	`Put/get_with_triggering_debug_callbacks`	Debug	No

For each variant, there are also different signatures available:

Table 2-17 Different Signatures of Each Variant

Style	Arguments
TLM2	`address`, `dataPtr`, `data_length`, `byte_enablePtr`, `enableLength`

Table 2-17 Different Signatures of Each Variant

Style	Arguments
TLM2-Word	`address`, `dataPtr`, `data_length`
Word	`DT`
Sub-Word	`DT`, `size`, `offset`

The following table describes the arguments mentioned in the table 2-16.

Table 2-18 Argument Description

Argument Name	C++	Description
`address`	`unsigned long long address`	Is the byte address as in TLM2 GP.
`dataPtr`	`(const) unsigned char* data`	Is the data array as in TLM2 GP.
`data_length`	`unsigned int dataLength`	Is the length of the transaction in bytes as in TLM2 GP.
`byte_enablePtr`	`const unsigned char* byteEneblePtr`	Specifies a byte enable array which can be 0 as in TLM2 GP.
`enableLength`	`unsigned int byteEnableLength`	Specifies the length of the byte enable array in bytes as in TLM2 GP.
`DT`	`(const) DT& data`	Specifies the data.
`size`	`unsigned int size`	Is the size of the access in bytes.
`offset`	`unsigned int offset`	Is the offset for the access in bytes.

Additional notes:

- The Trigger-Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the read data instead of passing it as an argument.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be `0`) since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives 2x16 different access methods that are supported by the memory objects, summarized in the following table:

Table 2-19 Supported Access Methods

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Simple	x	x	x	x
Simple-Debug	x	x	x	x
Trigger-Callbacks	x	x	x	x

Table 2-19 Supported Access Methods

Style	TLM2	TLM2-Word	Word	Sub-Word
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments, see the include file of the `register: scml2/reg.h`.

2.5.4.4 Triggering Callbacks on Bitfields

The register also has a set of API's to trigger the callbacks on its bitfields. These API's can be used for example, from within a callback.

API's to trigger bitfield behavior from register callbacks

```
bool put_with_triggering_bitfield_callbacks(const DT& data, sc_core::sc_time& t)
bool put_with_triggering_bitfield_callbacks(const DT& data, const DT& bitMask,
                                            sc_core::sc_time& t)
bool get_with_triggering_bitfield_callbacks(DT& data, sc_core::sc_time& t)
bool get_with_triggering_bitfield_callbacks(DT& data, const DT& bitMask,
                                            sc_core::sc_time& t)
bool put_debug_with_triggering_bitfield_callbacks(const DT& data, const DT& bitMask)
bool put_debug_with_triggering_bitfield_callbacks(const DT& data)
bool get_debug_with_triggering_bitfield_callbacks(DT& data)
bool get_debug_with_triggering_bitfield_callbacks(DT& data, const DT& bitMask)
```

2.5.4.5 Operators

The following assignment operators are available:

```
iterator begin()
const_iterator begin() const
```

Returns a random access iterator pointing to the `reg` object.

```
iterator end()
const_iterator end() const
```

Returns a random access iterator pointing to the end of the `reg` object.

A `reg` object can be converted to the underlying data type:

```
operator DT() const
```

The following assignment operators are available:

```
reg& operator=(DT value)
reg& operator =(const reg& r)
```

The following arithmetic assignment operators are available and behave as defined for the underlying data type:

```
reg& operator+=(DT value)
reg& operator-=(DT value)
reg& operator/=(DT value)
reg& operator*=(DT value)
reg& operator%=(DT value)
reg& operator^=(DT value)
reg& operator&=(DT value)
```

```
reg& operator|=(DT value)
reg& operator>>=(DT value)
reg& operator<<=(DT value)
```

The following prefix and postfix decrement and increment operators are available:

```
reg& operator--()
DT operator--(int)
reg& operator++()
DT operator++(int)
```

The register object has all the same properties and access methods as the `memory_alias` (name, parent, offset, index operator, callback overruling). The key additional feature for the register is that it can be used as a regular variable.

```
operator DT() const;
reg& operator=(DT value);
reg& operator =(const reg& r);
reg& operator+=(DT value);
reg& operator-=(DT value);
reg& operator/=(DT value);
reg& operator*=(DT value);
reg& operator%=(DT value);
reg& operator^=(DT value);
reg& operator&=(DT value);
reg& operator|=(DT value);
reg& operator>>=(DT value);
reg& operator<<=(DT value);
reg& operator--();
DT operator--(int);
reg& operator++();
DT operator++(int);
```

2.5.5 Callbacks

The `reg` object supports the callbacks listed below. For more information on callbacks, see also "Callbacks" on page 19.

2.5.5.1 Regular Callbacks

Every callback registration function accepts a few signatures for the callback methods:

Regular callback registration

```
set_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_read_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_write_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_word_read_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_word_write_callback(reg, SCML2_CALLBACK(method), syncType, tag)
set_post_write_callback(reg, SCML2_CALLBACK(method), syncType, tag);
```

The `set_callback` registration function accepts callback methods with the following signatures:

```
void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int tag)
void transportCallback(tlm::tlm_generic_payload&, int tag)
```

The `set_read_callback` registration function accepts callback methods with the following signatures:

```
bool readCallback(DT& data, const DT& byteEnables, sc_core::sc_time&, int tag)
bool readCallback(DT& data, const DT& byteEnables, int tag)
bool readCallback(DT& data, const DT& byteEnables, sc_core::sc_time&,
                  const scml2::tlm2_gp_extensions& extensions int tag)
bool readCallback(DT& data, const DT& byteEnables,
                  const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_write_callback` registration function accepts callback methods with the following signatures:

```
bool writeCallback(const DT& data, const DT& byteEnables, sc_core::sc_time&, int tag)
bool writeCallback(const DT& data, const DT& byteEnables, int tag)
bool writeCallback(const DT& data, const DT& byteEnables, sc_core::sc_time&,
                   const scml2::tlm2_gp_extensions& extensions int tag)
bool writeCallback(const DT& data, const DT& byteEnables,
                   const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_word_read_callback` registration function accepts callback methods with the following signatures:

```
bool wordReadCallback(DT& data, sc_core::sc_time&, int tag)
bool wordReadCallback(DT& data, int tag)
bool wordReadCallback(DT& data, sc_core::sc_time&,
                      const scml2::tlm2_gp_extensions& extensions int tag)
bool wordReadCallback(DT& data, const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_word_write_callback` registration function accepts callback methods with the following signatures:

```
bool wordWriteCallback(const DT& data, sc_core::sc_time&, int tag)
bool wordWriteCallback(const DT& data, int tag)
bool wordWriteCallback(const DT& data, sc_core::sc_time&,
                       const scml2::tlm2_gp_extensions& extensions int tag)
bool wordWriteCallback(const DT& data,
                       const scml2::tlm2_gp_extensions& extensions int tag)
```

The `set_post_write_callback` registration function accepts callback methods with the following signature:

```
void postWriteCallback(int tag)
```

Additional notes:

- The `bool` return value indicates whether the access was successful. The transport style callbacks should use the TLM2 response status.
- In all the above callback methods, *`tag`* is an optional argument, only to be used when the callback is registered with a tag.

- Streaming burst accesses are unrolled into word accesses and subword accesses are converted into word accesses with byte enables.
- The `byteEnables` mask will contain `0xff` for enabled bytes and `0x0` for disabled bytes.
- The `wordReadCallback` and `wordWriteCallback` types are for word accesses only. For unaligned accesses or subword accesses, an error response is returned.
- The untimed callbacks (without the `sc_time` parameter) cannot be `SELF_SYNCING` callbacks.

2.5.5.2 Debug Callbacks

The `reg` object support the following registration API's for debug callbacks.

Debug callback registration

```
set_debug_callback(reg, SCML2_CALLBACK(method), tag)
set_debug_read_callback(reg, SCML2_CALLBACK(method), tag)
set_debug_write_callback(reg, SCML2_CALLBACK(method), tag)
```

For debug callbacks, callback must have one of the following signatures:

```
unsigned int transportCallback(tlm::tlm_generic_payload&)
unsigned int transportCallback(tlm::tlm_generic_payload&, int tag)
```

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, `0` must be returned.

2.5.5.3 Predefined Callbacks

The following functions are available to register specific read and write behavior to a register.

`set_clear_on_read(reg)`	Clears all bits of the memory when the memory is read.
`set_set_on_read(reg)`	Sets all bits of the memory when the memory is read.
`set_clear_on_write_0(reg)`	Clears all bits to which the bit `0` is written.
`set_clear_on_write_1(reg)`	Clears all bits to which the bit `1` is written.
`set_write_once(reg)`	Sets all bits to which the bit `0` is written.
`set_set_on_write_0(reg)`	Sets all bits to which the bit `0` is written.
`set_set_on_write_1(reg)`	Sets all bits to which the bit `1` is written.
`set_write_once_error(reg)`	Allows a value to be written to the memory once. All subsequent writes will return an error. See the note text given below.
`set_write_once_ignore(reg)`	Allows a value to be written to the memory once. All subsequent writes will be ignored. See the note text given below.

The `write_once` callback function variants return a reference counted object of type `scml2::write_once_state`. This object can be used to reset the state, such that the memory becomes writable again, by calling `reset()` on it.

2.5.6 Access Restrictions

The `register` object supports access restrictions as described in "Access Restrictions" on page 21. The register supports all predefined access restrictions as specified in "Predefined Access Restrictions" on page 24, it supports both styles of access restriction callbacks:

```
scml2::access_restriction_result MyRestrictFtion(DT& data, DT& bit_enables, int tag)
scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& trans, int tag)
```

2.6 bitfield

The `bitfield` object is intended to access subword ranges in a register. It does not have its own storage, and must have a register as parent object. Bitfields are slightly different from the other storage objects, since they can be smaller than the `8-bit` minimal access size of a TLM2 transport access. Like a register, they cannot have any other aliases or registers; or by bound to the TLM2 target sockets. They also have a limited set of callbacks, only the `read` and `write` callbacks are allowed (no `transport` callbacks or callbacks with byte-enables and so on). Moreover, they have a limited list of access methods - only the ones with `datatype` as argument.

Other properties of `bitfield` object are:

- The bitfield can only be constructed with a register as parent. It has *`name`*, *`offset`*, and *`size`* parameters.
- It is possible to use it as a regular variable like a register, and all operators are overloaded.
- When a callback is registered with a bitfield, this will override the behavior of the parent register.
- When both the register as well as the bitfield have a callback registered, the register callback will be triggered. The rule of *most refined behavior* stops at the register. The register has additional APIs to call the bitfield callbacks from within the register callback implementation.
- `bitfield` objects can be attached to `reg` objects or other `bitfield` objects, to alias some of the bits in the parent object.
- The include file of the `bitfield` objects is `scml2/bitfield.h`.

2.6.1 Types

The bitfield class is templated with the underlying value type:

```
Template <typename DT> class bitfield
```

The following types are supported:

Table 2-20 Supported Datatypes

Supported Datatypes	Word Size
`unsigned char`	8
`unsigned short`	16
`unsigned int`	32
`unsigned long long`	64
`sc_dt::sc_biguint<128>`	128
`sc_dt::sc_biguint<256>`	256

Table 2-20 Supported Datatypes

Supported Datatypes	Word Size
`sc_dt::sc_biguint<512>`	512

When instantiating a `bitfield` object, it should have the same template value as its parent register.

2.6.2 Constructors

The following constructors are available:

```
bitfield(const std::string& name,
               reg<DT>& reg,
               unsigned int offset,
               unsigned int size)
bitfield(const std::string& name,
               bitfield<DT>& b,
               unsigned int offset,
               unsigned int size)
```

Creates a new bitfield. The *`offset`* and *`size`* arguments must be specified in bits.

2.6.3 Properties

The following methods are available to access the properties of a `bitfield`:

- `const std::string& get_name() const`

  Returns the full hierarchical name of the of the `bitfield` object.

- `unsigned long long get_offset() const`

  Returns the offset (in bits) of the `bitfield` start bit in the register.

- `unsigned long long get_size() const`

  Returns the size of the `bitfield` object in bits.

- `reg<DT>* get_register() const`

  Returns a pointer to the parent register object.

- `const std::string& get_description() const`
  `void set_description(const std::string&)`

  Gets/sets the description for the `bitfield` object. This description can be displayed in debuggers.

-
  ```
  bool has_default_read_behavior() const;
  bool has_default_write_behavior() const;
  bool has_default_debug_read_behavior() const;
  bool has_default_debug_write_behavior() const;
  bool has_never_syncing_read_behavior() const;
  bool has_never_syncing_write_behavior() const;
  bool is_dmi_read_allowed() const;
  bool is_dmi_write_allowed() const;
  ```

  These are a set of APIs to query what type of behavior is associated with the `memory_alias` and whether DMI is enabled.

- `bool has_default_read_restriction() const;`
  `bool has_default_write_restriction() const;`

  These APIs query whether there is a restriction set for the bitfield.

2.6.4 Behaviors

2.6.4.1 Put/Get Access

The `bitfield` object supports a limited set of the different `put/get` behaviors as described in "Behavior" on page 18. The set of access functions supported according to the different types and styles defined in "Behavior" on page 18 are detailed in the following table:

Table 2-21 Access Functions

Type	API	Callback	Watchpoints
Simple	`Put/get`	No	Yes
Simple-Debug	`Put/get_debug`	No	No
Trigger-Callbacks	`Put/get_with_triggering_callbacks`	`Regular`	Yes
Trigger-Debug-Callbacks	`Put/get_with_triggering_debug_callbacks`	`Debug`	No

For each variant, there are also different signatures available:

Table 2-22 Signatures Available for Access Functions

Style	Arguments
Bitfield access	`DT`

Additional notes:

- The Trigger-Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `get()` and `get_debug()` methods for word or subword accesses return the `read` data, instead of passing it as an argument.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives `2x4` different access methods that are supported by the `memory` objects, summarized in the following table:

Table 2-23 Access Methods Supported by Memory Objects

Style	Bitfield Access
Type	
Simple	x
Simple-Debug	x
Trigger-Callbacks	x
Trigger-Debug-Callbacks	x

2.6.4.2 Operators

A `bitfield` object can be converted to the underlying data type:

```
operator DT() const
```

The following assignment operators are available:

```
bitfield&amp; operator=(DT value)
bitfield& operator =(const bitfield& b)
```

The following arithmetic assignment operators are available and behave as defined for the underlying data type:

```
bitfield& operator+=(DT value)
bitfield& operator-=(DT value)
bitfield& operator/=(DT value)
bitfield& operator*=(DT value)
bitfield& operator%=(DT value)
bitfield& operator^=(DT value)
bitfield& operator&=(DT value)
bitfield& operator|=(DT value)
bitfield& operator<<=(DT value)
bitfield& operator>>=(DT value)
```

The following prefix and postfix decrement and increment operators are available:

```
bitfield& operator--()
DT operator--(int)
bitfield& operator++()
DT operator++(int)
```

2.6.5 Callbacks

The `bitfield` object supports the callbacks listed in this section. For more information on callbacks, also see "Callbacks" on page 19.

2.6.5.1 Regular Callbacks

Every callback registration function accepts a few signatures for the callback methods:

Regular callback registration

- `set_read_callback(bitfield, SCML2_CALLBACK(method), syncType, tag)`
- `set_write_callback(bitfield, SCML2_CALLBACK(method), syncType, tag)`
- `set_post_write_callback(bitfield, SCML2_CALLBACK(method), syncType, tag);`

The `set_read_callback` registration function accepts callback methods with the following signatures:

- `bool readCallback(DT& value, sc_core::sc_time&, int tag)`
- `bool readCallback (DT& value, int tag)`
- `bool readCallback (DT& value, sc_core::sc_time&,`
  `const scml2::tlm2_gp_extensions& extensions int tag)`
- `bool readCallback (DT& value, const scml2::tlm2_gp_extensions& extensions int tag)`

The `set_write_callback` registration function accepts callback methods with the following signatures:

- `bool writeCallback(const DT& `***`value`***`, sc_core::sc_time&, int `***`tag`***`)`
- `bool writeCallback(const DT& `***`value`***`, int `***`tag`***`)`
- `bool writeCallback(const DT& `***`value`***`, sc_core::sc_time&,`
  `const scml2::tlm2_gp_extensions& `***`extensions`***` int `***`tag`***`)`
- `bool writeCallback(const DT& `***`value`***`,`
  `const scml2::tlm2_gp_extensions& `***`extensions`***` int `***`tag`***`)`

The `set_post_write_callback` registration function accepts callback methods with the following signature:

- `void postWriteCallback(int `***`tag`***`)`

Additional notes:

- The `bool` return value indicates that the access was successful.
- In all the above callback methods, *`tag`* is an optional argument, only to be used when the callback is registered with a tag.
- The untimed callbacks (without the `sc_time` parameter) cannot be `SELF_SYNCING` callbacks.

2.6.5.2 Debug Callbacks

The `bitfield` object supports the following registration APIs for debug callbacks:

Debug callback registration:

- `set_debug_callback(bitfield, SCML2_CALLBACK(method), `***`tag`***`)`
- `set_debug_read_callback(bitfield, SCML2_CALLBACK(method), `***`tag`***`)`
- `set_debug_write_callback(bitfield, SCML2_CALLBACK(method), `***`tag`***`)`

For debug callbacks, `callback` must have one of the following signatures:

- `bool readCallback(DT& `***`value`***`, int `***`tag`***`)`
- `bool writeCallback(const DT& `***`value`***`, int `***`tag`***`)`

The `bool` return indicates whether the debug access was successful.

2.6.5.3 Predefined Callbacks

The following functions are available to register-specific `read`, and `write` behavior to a register.

`set_clear_on_read(bitfield)`	Clears all bits of the memory when the memory is read.
`set_set_on_read(bitfield)`	Sets all bits of the memory when the memory is read.
`set_clear_on_write_0(bitfield)`	Clears all bits to which the bit `0` is written.
`set_clear_on_write_1(bitfield)`	Clears all bits to which the bit `1` is written.
`set_set_on_write_0(bitfield)`	Sets all bits to which the bit `0` is written.
`set_set_on_write_1(bitfield)`	Sets all bits to which the bit `1` is written.

The `write_once` callback function variants return a reference counted object of type `scml2::write_once_state`. This object can be used to reset the state, such that the memory becomes writable again, by calling `reset()` on it.

2.6.6 Access Restrictions

The `bitfield` object supports access restrictions as described in "Access Restrictions" on page 21. The bitfield supports all predefined access restrictions as specified in "Predefined Access Restrictions" on page 24, it supports both styles of access restriction callbacks:

- `scml2::access_restriction_result MyRestrictFtion(DT& *data*, DT& bit_enables, int *tag*)`
- `scml2::access_restriction_result MyRestrictFtion(tlm::tlm_generic_payload& *trans*, int *tag*)`

2.7 router

The `router` is a SCML2 storage object that is intended to be used to create a configurable model, where memory access calls can be redirected to different storage components. Examples of models that can be built using a `router` are caches, memory controller, even interconnect components, or a dynamic address decoder. A `router` is similar to an `scml2::memory`, so it can be instantiated as a top-level memory object, but it does not implement any storage. It also does not have a default behavior; it requires a transport callback to implement its behavior. The key feature is, however, the additional mapping interface which allows routing certain memory regions to a `scml2::memory`, or an initiator socket (or anything implementing the `mappable_if`). Once a region is mapped, all accesses to that region will automatically be forwarded to the memory or socket without being evaluated in the `router` again. Only when the region is unmapped by the `router`, the transport implementation will be called again.

The `include` file of the `router` objects is `scml2/router.h`.

Additional properties of the `router` object are:

- It is possible to have a different mapping for `reads` and `writes`.
- The `router` has a limited set of `put/get` access methods, basically the ones that are similar to the TLM2 APIs.
- The `map` API maps a region of a certain size starting at the base address in the `router`, to a region in the destination, implementing the `mappable_if` interface starting from `address` offset. The `bool` return value will indicate failure if the region is already mapped, out of range, or if the request is not word-aligned.
- A `router` object is similar to a `memory` object, but it has no associated storage and no default behavior. A callback must be registered to the `router` object that implements the desired behavior of the accesses to this memory range.
- A `router` object can map a memory region to a region into a `memory` object, another `router` object, a `tlm2_gp_initiator_adapter` object, or an object that implements the `mappable_if` interface. Accesses to mapped regions do not trigger the attached callback, but are automatically forwarded to the destination object.
- The `router` object implements the `mappable_if` object, which means that it can be the destination for a mapped range of a `router` object.
- A `router` object cannot have aliases and/or registers.

2.7.1 Types

The `router` class is templated with the underlying value type:

```
Template <typename DT> class router
```

The following types are supported:

Table 2-24 Supported Datatypes

Supported Datatypes	Word Size
`unsigned char`	`8`
`unsigned short`	`16`
`unsigned int`	`32`
`unsigned long long`	`64`
`sc_dt::sc_biguint<128>`	`128`
`sc_dt::sc_biguint<256>`	`256`
`sc_dt::sc_biguint<512>`	`512`

2.7.2 Constructors

The following constructor is available:

`router(const std::string& name, unsigned long long size)`

Creates a new `router`. The `size` argument must be specified in words.

2.7.3 Properties

The following methods are available to access the properties of a `memory_alias`:

- `const std::string& get_name() const`

  Returns the full hierarchical name of the of the `router` object.
- `unsigned long long get_offset() const`

  Always returns `0` for a `router` object.
- `unsigned long long get_size() const`

  Returns the size of the `router` object in words.
- `unsigned int get_width() const`

  Returns the data width of the `router` object in bytes.
- `bool is_dmi_enabled()`

  Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.
- `void enable_dmi()`
  `void disable_dmi()`

  Enables/disables DMI accesses for the object.
- `const std::string& get_description() const`
  `void set_description(const std::string&)`

  Gets/sets the description for the `memory_alias` object. This description can be displayed in debuggers.

The `router` object can be bound to a TLM2 target socket via a `port_adapter` (see "Port Adaptors" on page 77). A target port adaptor may be bound to an SCML2 `router` as shown below.

```
// bind adaptor to memory
scml2::router my_router("my_ router ", 0x100);
(*my_port_adapter)(my_ router);
```

2.7.4 Behaviors

2.7.4.1 Mapping APIs

The following methods are available to map or unmap memory regions:

-
  ```
  bool map(unsigned long long base,
           unsigned long long size,
           mappable_if& destination,
           unsigned long long offset);
  ```

  Maps the memory range `[base, base + size]` of the socket to which the `router` object is bound to the memory range `[offset, offset + size]` of the destination.

  Possible destinations are: `memory`, `router`, `tlm2_gp_initiator_adapter`, and objects implementing the `mappable_if` interface.

  The *`base`*, *`size`*, and *`offset`* arguments must all be specified in bytes.

  If the mapping succeeds, `true` is returned; otherwise `false` is returned. The mapping will fail if:

  - the mapped range overlaps with a previously mapped range, or
  - if the mapped range is outside the memory range of the `router` object, or
  - if the base address or size of the mapped range is not aligned with the width of the `router` object.

  After mapping a region to a destination, all accesses coming to this region are automatically forwarded to this destination. If a callback is registered to `scml_router`, it is not invoked. If a burst goes across the boundary of a mapped region, then the burst is unrolled.

  The `map()` method maps the memory range both for `read` access and `write` accesses.

-
  ```
  bool map_read(unsigned long long base,
                unsigned long long size,
                mappable_if& destination,
                unsigned long long offset);
  ```

  Same as the `map()` method, but the memory range is mapped only for `read` accesses.

-
  ```
  bool map_write(unsigned long long base,
                 unsigned long long size,
                 mappable_if& destination,
                 unsigned long long offset);
  ```

  Same as the `map()` method, but the memory range is mapped only for `write` accesses.

  Note Mapped ranges for `read` and for `write` accesses are completely independent. A mapped range for `read` accesses is allowed to overlap with a mapped range for `write` accesses.

- `bool unmap(unsigned long long` *`base`*`);`

  Unmaps a previously mapped range for both `read` and `write` accesses.

  Returns `true` if a mapped range is found and removed, otherwise `false`.

- `bool unmap_read(unsigned long long` *`base`*`);`

Same as the `unmap()` method, but the memory range is unmapped only for `read` accesses.

- `bool unmap_write(unsigned long long` *`base`*`);`

  Same as the `unmap()` method, but the memory range is unmapped only for `write` accesses.

- `void unmap_all();`

  Unmaps all previously mapped memory regions.

Note Mapping memory regions can be done:
- statically from the constructor of the module,
- or dynamically from the attached callback or from another SystemC thread.

2.7.4.2 Transport Calls

The `router` object implements the following TLM2 methods:

- `void b_transport(tlm::tlm_generic_payload&` *`trans`*`, sc_core::sc_time&` *`t`*`)`
  `unsigned int transport_dbg(tlm::tlm_generic_payload&` *`trans`*`)`
- `bool get_direct_mem_ptr(tlm::tlm_generic_payload&` *`trans`*`, tlm::tlm_dmi&` *`dmiData`*`)`

These methods trigger the callbacks registered to the `router` object.

2.7.4.3 Put/Get access

The `router` object supports some of the `put/get` behaviors as described in "Behavior" on page 18. So, only access methods that trigger callbacks are supported. The following set of access functions is supported according to the different types and styles as defined in "Behavior" on page 18:

Table 2-25 Access Functions

Type	API	Callback	Watchpoints
Trigger-Callbacks	`Put/get_with_triggering_callbacks`	`Regular`	Yes
Trigger-Debug-Callbacks	`Put/get_with_triggering_debug_callbacks`	`Debug`	No

For each variant, there are also different signatures available:

Table 2-26 Signatures Available for Access Functions

Style	Arguments
TLM2	`address`, `dataPtr`, `data_length`, `byte_enablePtr`, `enableLength`
TLM2-Word	`address`, `dataPtr`, `data_length`
Word	`index`, `DT`
Sub-Word	`index`, `DT`, `size`, `offset`

In the table 2-26, the arguments are as follows:

Table 2-27 Arguments for Signatures of Access Functions

Argument Name	C++	Description
`address`	`unsigned long long address`	Specifies the byte address as in TLM2 GP.

Table 2-27 Arguments for Signatures of Access Functions

Argument Name	C++	Description
`dataPtr`	`(const) unsigned char* data`	Specifies the data array as in TLM2 GP.
`data_length`	`unsigned int dataLength`	Specifies the length of the transaction in bytes as in TLM2 GP.
`byte_enablePtr`	`const unsigned char* byteEneblePtr`	Specifies a byte enable array which can be `0` as in TLM2 GP.
`enableLength`	`unsigned int byteEnableLength`	Specifies the length of the byte enable array in bytes as in TLM2 GP.
`index`	`unsigned long long index`	Specifies the word index as specified by the DT template of the `memory` object.
`DT`	`(const) DT& data`	Specifies the data.
`size`	`unsigned int size`	Specifies the size of the access in bytes.
`offset`	`unsigned int offset`	Specifies the offset for the access in bytes.

Additional notes:

- The Trigger- Callback access functions take an additional `sc_time` argument. They return the TLM2 response status, returned by the triggered callback.
- The `put_debug_with_triggering_callbacks()` and `get_debug_with_triggering_callbacks()` calls must not use `byte_enables` (the `byteEnablePtr` must be `0`), since TLM2 does not support byte enables for debug calls.
- The debug versions of these methods do not trigger watchpoints.

In total, this gives `2x8` different access methods that are supported by the `memory` objects, summarized in the following table:

Table 2-28 Access Methods Supported by Memory Objects

Style	TLM2	TLM2-Word	Word	Sub-Word
Type				
Trigger-Callbacks	x	x	x	x
Trigger-Debug-Callbacks	x	x	x	x

For a complete list with all arguments, see the `include` file of the memory: `scml2/router.h`.

2.7.5 Callbacks

The `router` object supports the callbacks listed in this section. For more information on callbacks, also see "Callbacks" on page 19.

2.7.5.1 Regular Callbacks

The `router` object only supports the `set_callback` registration-style callback methods.

Regular callback registration:

- `set_callback(mem, SCML2_CALLBACK(method), *syncType*, *tag*)`

It only accepts callback methods with the following signature:

- `void transportCallback(tlm::tlm_generic_payload&, sc_core::sc_time&, int *tag*)`

Note In all the above callback methods, `tag` is an optional argument, only to be used when the callback is registered with a tag.

2.7.5.2 Debug Callbacks

The `memory` object supports the following registration APIs for debug callbacks:

Debug callback registration

- `set_debug_callback(mem, SCML2_CALLBACK(method), *tag*)`

For debug callbacks, `method` must have one of the following signatures:

- `unsigned int transportCallback(tlm::tlm_generic_payload&)`
- `unsigned int transportCallback(tlm::tlm_generic_payload&, int *tag*)`

The return value is the number of consecutive bytes successfully read or written. If the access cannot be executed, `0` must be returned.

2.8 memory utilities

This section describes:

- memory_index_reference
- mappable_if
- Callback Base Classes
- Convenience Functions

2.8.1 memory_index_reference

The `memory_index_reference` object is returned by the `lvalue` version (non-`const` version) of the index operator (`operator[]`) of `memory` and `memory_alias` objects.

The `memory_index_reference` object forwards all operations to the referenced `memory` object.

The include file of the `memory_index_reference` objects is `scml2/memory_index_reference.h`.

The following sections describe:

- Types
- Access Methods
- Operators

2.8.1.1 Types

The following type definitions are available:

```
typedef DT data_type
typedef memory_index_reference<DT> reference
```

2.8.1.2 Access Methods

The following access methods are available:

```
void put(const DT& value)
DT get() const

void put_debug(const DT& value)
DT get_debug() const
```

2.8.1.3 Operators

A `memory_index_reference` object can be converted to the underlying data type of the referenced `memory` object:

```
operator DT() const
```

The following assignment operators are available:

```
reference& operator=(DT value)
```

The following arithmetic assignment operators are available and behave as defined for the underlying data type of the referenced `memory` object:

```
reference& operator+=(DT value)
reference& operator-=(DT value)
reference& operator/=(DT value)
reference& operator*=(DT value)
reference& operator%=(DT value)
reference& operator^=(DT value)
reference& operator&=(DT value)
reference& operator|=(DT value)
reference& operator>>=(DT value)
reference& operator<<=(DT value)
```

The following prefix and postfix decrement and increment operators are available:

```
reference& operator--()
DT operator--(int)
reference& operator++()
DT operator++(int)
```

2.8.2 mappable_if

The `mappable_if` object is the abstract interface that must be implemented by an object to be able to act as a destination for a mapped range of a `router` object.

The include file of the `mappable_if` objects is `scml2/mappable_if.h`.

The following section describes:

- TLM API Methods

2.8.2.1 TLM API Methods

The following methods must be implemented:

```
std::string get_mapped_name() const = 0
```

Should return the name of the mapped destination. For a `memory` or `router` object, this is the name of the object. For a `tlm2_gp_target_adapter` object, this is the name of the TLM2 initiator socket.

```
void register_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface) = 0
void unregister_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface) = 0
```

Is called to register/unregister a pointer to a `tlm_bw_direct_mem_if` object. When the object that inherits from the `mappable_if` has to invalidate the DMI pointers, it has to call `invalidate_direct_mem_ptr()` on each registered interface.

Note If a `tlm_bw_direct_mem_if` object is registered multiple times, it must only be stored once and the invalidate call must only be called once.

The following TLM2 API methods (see the *IEEE Std 1666 TLM-2.0 Language Reference Manual*) must be implemented:

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) = 0
bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmiData)
                                                                      = 0
unsigned int transport_dbg(tlm::tlm_generic_payload& trans) = 0
```

2.8.3 Callback Base Classes

This section describes:

- memory_callback_base
- memory_debug_callback_base
- router_callback_base
- router_debug_callback_base
- bitfield_read_callback_base
- bitfield_write_callback_base
- bitfield_debug_read_callback_base
- bitfield_debug_write_callback_base

2.8.3.1 memory_callback_base

Base class for regular callbacks of `memory`, `memory_alias`, or `reg` objects.

The following virtual methods must be implemented:

```
void execute(tlm::tlm_generic_payload& trans, sc_core::sc_time& t) = 0
```

Implementation of the callback behavior.

```
bool has_never_syncing_behavior() const = 0
```

Returns `true` if the callback never synchronizes, otherwise `false`.

The include file of the `memory_callback_base` objects is `scml2/memory_callback_base.h`.

2.8.3.2 memory_debug_callback_base

Base class for debug callbacks of `memory`, `memory_alias`, or `reg` objects.

The following virtual method must be implemented:

```
unsigned int execute(tlm::tlm_generic_payload& trans) = 0
```

Implementation of the callback behavior.

The include file of the `memory_debug_callback_base` objects is `scml2/memory_debug_callback_base.h`.

2.8.3.3 router_callback_base

Base class for regular callbacks of `router` objects.

The following virtual methods must be implemented:

`void execute(tlm::tlm_generic_payload& trans, sc_core::sc_time& *t*) = 0`

Implementation of the callback behavior.

`bool has_never_syncing_behavior() const = 0`

Returns `true` if the callback never synchronizes, otherwise `false`.

The include file of the `router_callback_base` objects is `scml2/router_callback_base.h`.

2.8.3.4 router_debug_callback_base

Base class for debug callbacks of `router` objects.

The following virtual method must be implemented:

`unsigned int execute(tlm::tlm_generic_payload& *trans*) = 0`

Implementation of the callback behavior.

The include file of the `router_debug_callback_base` objects is `scml2/router_debug_callback_base.h`.

2.8.3.5 bitfield_read_callback_base

Templated base class for regular read callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual methods must be implemented:

`bool read(DT& value, sc_core::sc_time& *t*) = 0`

Implementation of the callback behavior.

`bool has_never_syncing_behavior() const = 0`

Returns `true` if the callback never synchronizes, otherwise `false`.

The include file of the `bitfield_read_callback_base` objects is `scml2/bitfield_read_callback_base.h`.

2.8.3.6 bitfield_write_callback_base

Templated base class for regular write callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual methods must be implemented:

`bool write(const DT& value, sc_core::sc_time& *t*) = 0`

Implementation of the callback behavior.

`bool has_never_syncing_behavior() const = 0`

Returns `true` if the callback never synchronizes, otherwise `false`.

The include file of the `bitfield_write_callback_base` objects is `scml2/bitfield_write_callback_base.h`.

2.8.3.7 bitfield_debug_read_callback_base

Templated base class for debug read callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual method must be implemented:

```
bool read(DT& value) = 0
```

Implementation of the callback behavior.

The include file of the `bitfield_debug_read_callback_base` objects is `scml2/bitfield_debug_read_callback_base.h`.

2.8.3.8 bitfield_debug_write_callback_base

Templated base class for debug write callbacks of `bitfield` objects. The class has one template parameter, which is the data type of the `bitfield` object.

The following virtual method must be implemented:

```
bool write(const DT& value) = 0
```

Implementation of the callback behavior.

The include file of the `bitfield_debug_write_callback_base` objects is `scml2/bitfield_debug_write_callback_base.h`.

2.8.4 Convenience Functions

The following convenience functions are available in `scml2/utils.h`:

```
template <typename DT> DT extract_bits(const DT& v, unsigned int sizeBits,
                                                  unsigned int offsetBits)
```

Returns `sizeBits` bits from offset `offsetBits` of the data word `v`. `offsetBits` and `sizeBits` are specified in bits.

Little endian bit ordering is used (the offset of the `lsb` is `0).`

```
template <typename DT> DT insert_bits(const DT& v, const DT& rhs,
                    unsigned int sizeBits, unsigned int offsetBits)
```

Inserts `sizeBits` bits of the data passed in `rhs` at offset `offsetBits` in the data word `v` and returns the result. `offsetBits` and `sizeBits` are specified in bits.

Little endian bit ordering is used (the offset of the `lsb` is `0`).

2.9 Deprecated API's and Adapters

2.9.1 Callbacks

The following convenience callback functions are deprecated:

2.9.1.1 Disallow Access to Memory Object

The following functions are available to disallow the access to a memory object. These callbacks register a callback of type `memory_disallow_access_callback`.

Predefined API's to disallow access to a memory object

```
set_ignore_access(mem)
set_ignore_read_access(mem)
set_ignore_write_access(mem)
set_disallow_access(mem)
set_disallow_read_access(mem)
set_disallow_write_access(mem)
set_read_only(mem)
set_write_only(mem)
```

Where `mem` is the memory object to which the callback will be registered.

When an access is ignored, an *ok* response is returned; when an access is disallowed, an error response is returned. Reading a write-only memory or writing a read-only memory will also return an error response.

When an access is ignored or disallowed, the contents of the memory is not updated after the access.

The following functions are available to disallow the debug access to a memory. These callbacks register a debug callback of type `memory_disallow_debug_access_callback`. The callback ignores the access and returns `0`. When an access is ignored or disallowed, the contents of the memory is not updated after the access.

Predefined API's to disallow debug accesses to a memory object

```
set_disallow_debug_access(mem)
set_disallow_debug_read_access(mem)
set_disallow_debug_write_access(mem)
```

where `mem` is the memory object to which the callback will be registered. When an access is ignored or disallowed, the contents of the memory is not updated after the access.

2.9.1.2 Callbacks on Predefined Behaviors

The following functions are available to register a user-defined callback, in combination with the predefined behaviors, where the user callback is called before the behavior callback:

Pre predefined callback registration

```
set_clear_on_read_callback(mem, object, callback, name, syncType)
set_clear_on_read_callback(mem, object, callback, name, syncType, tag)
set_word_clear_on_read_callback(mem, object, callback, name, syncType)
set_word_clear_on_read_callback(mem, object, callback, name, syncType, tag)
set_set_on_read_callback(mem, object, callback, name, syncType)
set_set_on_read_callback(mem, object, callback, name, syncType, tag)
set_word_set_on_read_callback(mem, object, callback, name, syncType)
set_word_set_on_read_callback(mem, object, callback, name, syncType, tag)
set_clear_on_write_0_callback(mem, object, callback, name, syncType)
set_clear_on_write_0_callback(mem, object, callback, name, syncType, tag)
set_word_clear_on_write_0_callback(mem, object, callback, name, syncType)
set_word_clear_on_write_0_callback(mem, object, callback, name, syncType, tag)
set_clear_on_write_1_callback(mem, object, callback, name, syncType)
set_clear_on_write_1_callback(mem, object, callback, name, syncType, tag)
set_word_clear_on_write_1_callback(mem, object, callback, name, syncType)
set_word_clear_on_write_1_callback(mem, object, callback, name, syncType, tag)
set_write_once_ignore_callback(mem, object, callback, name, syncType)
set_write_once_ignore_callback(mem, object, callback, name, syncType, tag)
set_write_once_error_callback(mem, object, callback, name, syncType)
set_write_once_error_callback(mem, object, callback, name, syncType, tag)
```

```
set_word_write_once_ignore_callback(mem, object, callback, name, syncType)
set_word_write_once_ignore_callback(mem, object, callback, name, syncType, tag)
set_word_write_once_error_callback(mem, object, callback, name, syncType)
set_word_write_once_error_callback(mem, object, callback, name, syncType, tag)
set_set_on_write_0_callback(mem, object, callback, name, syncType)
set_set_on_write_1_callback(mem, object, callback, name, syncType)
set_set_on_write_1_callback(mem, object, callback, name, syncType, tag)
set_word_set_on_write_0_callback(mem, object, callback, name, syncType)
set_word_set_on_write_1_callback(mem, object, callback, name, syncType)
set_word_set_on_write_1_callback(mem, object, callback, name, syncType, tag)
```

The following functions are available to register a user-defined callback, in combination with the above defined behaviors, where the user callback is called after the behavior callback:

Post predefined callback registration

```
set_post_clear_on_write_0_callback(mem, object, callback, name, syncType)
set_post_clear_on_write_0_callback(mem, object, callback, name, syncType, tag)
set_post_clear_on_write_1_callback(mem, object, callback, name, syncType)
set_post_clear_on_write_1_callback(mem, object, callback, name, syncType, tag)
set_post_write_once_ignore_callback(mem, object, callback, name, syncType)
set_post_write_once_ignore_callback(mem, object, callback, name, syncType, tag)
set_post_write_once_error_callback(mem, object, callback, name, syncType)
set_post_write_once_error_callback(mem, object, callback, name, syncType, tag)
set_post_set_on_write_0_callback(mem, object, callback, name, syncType)
set_post_set_on_write_1_callback(mem, object, callback, name, syncType)
set_post_set_on_write_1_callback(mem, object, callback, name, syncType, tag)
```

Where:

mem	Is the memory object to which the callback will be registered.
object	Is a pointer to the class containing the callback method.

callback	Is a pointer to a member function of the object class. It must have one of the following signatures:
	For the regular callbacks, it should have one of the `transportCallback`, `readCallback`, or `wordReadCallback` signatures, as listed above.
	For the post callbacks, it should be either:
	`void postWriteCallback()`
	`void postWriteCallback(int tag)`

name	Is a string specifying the name of the callback function.

syncType	Can be one of the following: • NEVER_SYNCING indicates that the callback is nonblocking and must never call wait(). • SELF_SYNCING indicates that the callback is blocking and may call wait(). The timing annotation is passed unmodified to the callback. • AUTO_SYNCING indicates that the callback is blocking and may call wait(). The memory object synchronizes before calling the callback. The timing annotation passed to the callback is always SC_ZERO_TIME. These types are defined in the scml2/types.h file. The Post predefined behavior callbacks do not support SELF_SYNCING callbacks (since the callback does not have a time argument).
tag	Is a user-provided integer that is passed to the callback.

2.9.2 TLM2 Adapters

The tlm2_gp_target_adapter and tlm2_gp_initiator_adapter are deprecated in favor of the port_adapters described in Port Adaptors.

2.9.3 tlm2_gp_target_adapter

scml2::tlm2_gp_target_adapter is an adapter that is used to bind a memory object to a tlm2 target socket. It takes a TLM2 GP transaction and forwards it to the memory objects. The adapter takes care of burst accesses (burst unrolling so that the different regions are accessed correctly) it also takes care of the AT to LT conversion for the TLM2 base protocol. It ensures that all accesses to memory objects can be executed as LT accesses (b_transport semantics, that is, calling wait() and so on is allowed). The adapter does not touch any extensions so ignorable extensions are forwarded.

The adapter also implements the backward DMI interface, this means that the memories will use the target adapter to issue the invalidate DMI pointer calls to the initiator.

The tlm2_gp_target_adapter can be bound to any object implementing the mappable_if, for example, an scml2::memory.

The tlm2_gp_target_adapter is specific to the TLM2 base protocol, other protocol definitions may need their own adapter implementation to take care of specific burst unrolling features or for their specific AT to LT conversion. To create an adapter, it is required to create an object that implements the following:

- The tlm::tlm_fw_transport_if so that it can be bound to a target socket.
- A binding operation () with a mappable_if so that the storage objects can be bound to it.

The tlm2_gp_target_adapter object is used to bind an object that implements the mappable_if (for example, a memory or router object) to a tlm_target_socket.

All TLM2 API methods are forwarded to the object bound to the adapter.

The include file of the tlm2_gp_target_adapter objects is scml2/tlm2_gp_target_adapter.h.

```
template <unsigned int BUSWIDTH>
class tlm2_gp_target_adapter   :
   public sc_core::sc_object,
   public tlm::tlm_fw_transport_if<>,
   public tlm::tlm_bw_direct_mem_if
{
typedef tlm::tlm_base_target_socket<BUSWIDTH,
                    tlm::tlm_fw_transport_if<>,
                    tlm::tlm_bw_transport_if<>,
                    N,
                    POL> socket_type;

tlm2_gp_target_adapter(const std::string& name, socket_type& s);
void operator()(mappable_if& destination);
...
};
```

The following sections describe:

- Types
- Constructors
- Binding
- Custom Forwarding

2.9.3.1 Types

The `tlm2_gp_target_adapter` class is templated with the `BUSWIDTH`:

```
template <unsigned int BUSWIDTH> class tlm2_gp_target_adapter
```

The `BUSWIDTH` must be the same as the `BUSWIDTH` of the TLM2 target socket to which the adapter is bound.

2.9.3.2 Constructors

The following constructor is available:

```
tlm2_gp_target_adapter(const std::string& name,
                       tlm::tlm_base_target_socket<BUSWIDTH>& s)
```

Creates a target adapter and binds it to the TLM2 target socket.

2.9.3.3 Binding

The following method is available to bind objects that inherit from the `mappable_if` object to the `tlm2_gp_target_adapter` object:

```
void operator()(mappable_if& destination)
```

Binds an object to the adapter class.

2.9.3.4 Custom Forwarding

By default, `tlm2_gp_target_adapter` always forwards all TLM2 API methods to the first bound `mappable_if` object. It is possible to register a user-defined function to forward transactions to other bound `mappable_if` objects. This is done by calling `set_select_callback()`, and passing an SCML2 callback method which returns a `mappable_if` pointer for a given TLM payload.

For example:

```
MyModule(sc_module_name name) ...{
    ...
    adapter(memory1);
    adapter(memory2);
    set_select_callback(adapter, SCML2_CALLBACK(selectMemory));
    ...
}
```

```
scml2::mappable_if* selectMemory(tlm::tlm_generic_payload& trans) {
    if (...) {
      // Change the transaction address, and forward to memory 1
      trans.set_address(4);
      return &memory1;
    }
    else {
      // Change the transaction address, and forward to memory 2
      trans.set_address(8);
      return &memory2;
    }
}
```

2.9.4 tlm2_gp_initiator_adapter

`scml2::tlm2_gp_initiator_adaper` is an object that is used to map a memory region of a `router` object to a `tlm_initiator_socket`. It implements the `mappable_if`, so you can use it as a destination for the map API's of the `router` and bind it to an initiator socket.

The `tlm2_gp_initiator_adapter` object is used to map a memory region of a `router` object to a `tlm_initiator_socket`.

The `tlm2_gp_initiator_adapter` object binds to the `tlm_initiator_socket` and implements the `mappable_if`.

The include file of the `tlm2_gp_initiator_adapter` objects is `scml2/tlm2_gp_initiator_adapter.h`.

```
template <unsigned int BUSWIDTH, int N = 1,
                 sc_core::sc_port_policy POL = sc_core::SC_ONE_OR_MORE_BOUND>
class tlm2_gp_initiator_adapter   :
      public sc_core::sc_object,
      public mappable_if,
      public tlm::tlm_bw_transport_if
{
...
tlm2_gp_initiator_adapter(const std::string& name, socket_type& s);
...
};
```

The following sections describe:

- Types
- Constructors

2.9.4.1 Types

The `tlm2_gp_initiator_adapter` is templated with the `BUSWIDTH`:

```
template <unsigned int BUSWIDTH> class tlm2_gp_initiator_adapter
```

The `BUSWIDTH` must be the same as the `BUSWIDTH` of the TLM2 initiator socket to which the adapter is bound.

2.9.4.2 Constructors

The following constructor is available:

```
tlm2_gp_initiator_adapter(const std::string& name,
                          tlm::tlm_base_initiator_adapter<BUSWIDTH>& s)
```

Creates as initiator adapter and binds it to the TLM2 initiator socket.

Chapter 3
Clock Objects

This chapter describes the clock objects.

Note If the following error message is issued when simulating:

```
ERROR: Error in scml_clock 'divider': the attached master clock 'master' is no
scml_clock!
```

or

```
Error in get_scml_clock 'divider': the driving clock port must be bound to a channel
that implements scml_clock_if!
```

You must use an object of type `scml_clock` for the master clock object and export it using an `sc_export<sc_signal_inout_if<bool>> export`, as shown below.

```
sc_export<sc_signal_inout_if<bool>> p_CLK; // clock output port
scml_clock   m_clkObject; // master clock object
```

3.1 Overview

The following table summarizes the clock objects.

Table 3-1 Clock Objects

Modeling Object	Summary
`scml_clock`	It implements `sc_clock_if`. It is an optimized version of `sc_clock`.
`scml_divided_clock`	It is a clock derived from another clock by multiplying the start time and/or the period with specified integer factors.
`scml_clock_gate`	It is a module which takes a clock and an enable signal as inputs and produces a gated clock as output.
`scml_clock_counter`	It has to be attached to a clock and is used to get the number of clock edges that have happened in a certain period.
`scml2::clocked_module`	It is the base class for modules that want to receive SCML clock tick callbacks.

Table 3-1 Clock Objects

Modeling Object	Summary
`scml2::clocked_timer`	It is a modeling object that provides a timer callback mechanism based on an SCML clock.
`scml2::clocked_callback`	It is a convenience class that forwards a clock tick callback to any member function of a module without the need to inherit from the `clocked_module` base class.
`scml2::clocked_event`	It is a convenience class that allows a SystemC method or thread to wait until a certain clock tick happens.
`scml2::clocked_peq_container`	It is a modeling object for TLM2 models using the non-blocking APIs. It buffers payload arriving in the model, like multiple outstanding transactions, possibly coming with different timing annotations from different initiators.
`scml2::clocked_peq`	It is a modeling object similar to the `clocked_peq_container` that can trigger a callback whenever an element from the payload buffer becomes available.

3.2 Clocks and Reset

- scml_clock
- scml_divided_clock

3.2.1 scml_clock

An `scml_clock` object implements `sc_clock_if`.A number of extensions come with an `scml_clock` that are not available for an `sc_clock`:

- changing the period of a clock
- disabling/enabling a clock
- receiving a callback after a specified number of clock ticks

Objects of type `scml_clock` can be constructed using one of the following constructors:

```
scml_clock(const char* name,
           const sc_core::sc_time& period,
           double dutyCycle=0.5,
           const sc_core::sc_time& startTime=sc_core::SC_ZERO_TIME,
           bool posedgeFirst=true);

scml_clock(const char* name,
           double periodV,
           sc_core::sc_time_unit periodTu,
           double dutyCycle=0.5);

scml_clock(const char* name,
           double periodV,
           sc_core::sc_time_unit periodTu,
           double dutyCycle,
           double startTtimeV,
           sc_core::sc_time_unit startTimeTu,
           bool posedgeFirst=true);
```

where:

name	Specifies a name for the clock object.
period	Specifies the clock period.
periodV	Specifies the value of the clock period.
periodTu	Specifies the time unit for the clock period.
dutyCycle	Specifies the duty cycle of the clock object.
startTime	Specifies the time of the first clock edge.
startTimeV	Specifies the value of the time of the first clock edge.
startTimeTu	Specifies the time unit for the time of the first clock edge.
posedgeFirst	Specifies if the first edge will be a `posedge` or a *negedge*.

The following functions are available to set/get properties of the clock object:

```
const char* name() const;
```

Returns the name of the clock.

```
bool is_master() const;
```

Returns `true` for master clocks and `false` for divided (class `scml_divided_clock`) or gated clocks (class `scml_clock_gate`).

```
sc_core::sc_time get_period() const;
void set_period(const sc_core::sc_time &t);
```

Gets and sets the clock period. Setting the period is only valid for master clocks. It can be set (changed) at any time. The new value will be returned by `get_period()` after the next update phase of the SystemC kernel.

```
double get_duty_cycle()const;
void set_duty_cycle(double d);
```

Gets and sets the duty cycle. The value provided to `set_duty_cycle()` must be larger `0.0` and smaller `1.0`.

```
sc_core::sc_time& get_start_time()const;
void set_start_time(const sc_core::sc_time& t);
```

Gets or sets the start time of the clock. Immediately after changing the period or enabling the clock, it returns the start time of the next period, that is the first period following the new properties. Setting the start time is only valid for master clocks. It must only be called before the initialization phase of the SystemC kernel.

```
bool get_posedge_first()const;
void set_posedge_first(bool posedgeFirst);
```

Gets and sets the `posedge_first` property of the clock. It must only be called before the initialization phase of the SystemC kernel.

```
double get_period_multiplier()const;
void set_period_multiplier(double m);
```

Gets and sets the period multiplier of the clock. Setting the period multiplier is only valid for divided clocks. It can be set at any time. The new value will be returned by `get_period_multiplier()` after the next update phase of the SystemC kernel.

```
void enable();
void disable();
```

Enables (that is, makes active) or disables (that is, makes inactive) the clock. When an `scml_clock` is disabled, the output is `0` for a normal clock, and **1** for an inverted clock (starting with a negative edge, that is, `get_posedge_first()` returning `false`).

```
bool disabled();
```

Tests whether the clock is disabled.

```
bool running();
```

Tests whether the clock is running. For a master clock, this is equivalent to `!disabled()`. For a divided clock or gated clock, it also considers the running state of the master clock. That is, a divided clock or gated clock is running if it is enabled and the master clock is running.

The `scml_clock` provides a notification mechanism for changes of the parameters `period` and `enabled`. An observer of clock parameter changes must inherit the base class `scml_clock_observer` and implement the method `handle_clock_parameters_updated()`. When the period of a clock is changed, or a clock is enabled or disabled, the new parameters become active with the next update phase. During the update, the clock calls `handle_clock_parameters_updated()` for all registered observers. If multiple parameters of the clock are changed within the same cycle, then `handle_clock_parameters_updated()` is only called once.

```
void register_observer(scml_clock_observer* o);
void unregister_observer(scml_clock_observer* o);
```

Registers or unregisters a clock observer with the clock.

The `scml_clock` is the basic modeling object for the clocked modeling style of the SCML modeling style. It provides an ease of use mechanism to register callbacks with clock boundaries. A *clock boundary* is defined as the beginning of a period. For a normal clock, it corresponds to the positive edge; for an inverted clock (`get_posedge_first()` returning `false`) to a negative edge. Within this clock callback API, the clock boundary is called a *clock tick*.

Clock tick callbacks are called from the context of an SystemC method, that is internal to the clock. The call happens during the evaluation phase of the first delta cycle of the SystemC time, that corresponds to the clock edge.

Note SystemC processes that are sensitive to the clock edge event, will only be activated in the second delta cycle.

The main user interface of the clock tick mechanism is provided by the class `clocked_module`, see "scml2::clocked_module" on page 77. Within an `scml_clock`, the following functions are related to clock ticks:

```
bool check_at_tick() const;
```

Checks whether there is a clock tick at the current SystemC time. The result is independent from the current delta cycle. That means, `check_at_tick()` returns `true` already in the first delta cycle that corresponds to a clock tick, although processes that are sensitive to the clock edge event are only activated in the second delta cycle.

```
sc_core::sc_time get_next_edge_offset(bool pos_edge) const;
```

Returns the offset (as SystemC time) to the next positive or negative edge of the clock, depending on the value of the argument `pos_edge`.

```
unsigned long long get_tick_count() const;
```

Queries the tick count of the clock. This corresponds to the number of clock ticks, since the start of the simulation. The return value takes all phases when the clock was disabled, as well as, all changes of the clock period into account.

This function has been optimized for speed when called in a synchronous way, that is from a tick callback. An unsynchronized call is more expensive, but only the first time for a given SystemC time, due to internal caching of the result.

```
sc_dt::uint64 get_clock_count() const;
```

Returns the same value as `get_tick_count()`. This function is provided for backward compatibility.

```
unsigned long long get_tick_count(const sc_core::sc_time& delay) const;
```

Returns the tick count of the clock for the future SystemC time `sc_time_stamp()` + `delay`. For example, if called with `delay=SC_ZERO_TIME`, it returns the current tick count, which is equivalent to `get_tick_count()`.

```
sc_core::sc_time get_tick_time(long long clock_ticks_to_skip) const;
```

Gets the SystemC time for the clock tick that happens after `clock_ticks_to_skip` ticks from now. For example if called with `clock_ticks_to_skip=0`, it returns the time of the next tick.

```
void get_next_tick_data(const sc_core::sc_time& delay, unsigned long long& count,
sc_core::sc_time& time) const;
```

Gets the clock count and SystemC time for the next clock tick after a given future SystemC time. The future point in time is given by the `sc_time` argument `delay`, which is interpreted relative to the current SystemC time `sc_time_stamp()`. If called synchronized (the future point in time corresponds to a clock tick), it returns the data for the next following clock tick.

Several events are available for clock objects. They can be accessed using the following functions:

```
const sc_event& value_changed_event() const;
const sc_event& posedge_event() const;
const sc_event& negedge_event() const;

bool event();
bool posedge() const;
bool negedge() const;
```

Boolean functions to test whether a certain event occurred.

For tracing purposes, a reference to the current value can be obtained:

```
const bool & get_data_ref() const;
```

Note The `scml_clock` cannot be optimized when tracing is enabled. Enabling tracing will disable clock optimizations.

3.2.2 scml_divided_clock

A divided clock is a special version of an `scml_clock` that is derived from another clock by multiplying the start time and/or the period with specified integer factors. In case both multipliers are `1`, a local mirror of the clock is obtained. An advantage of such a mirror of a clock is that it can be enabled and disabled locally.

Objects of type `scml_divided_clock` can be constructed using one of the following constructors:

```
scml_divided_clock(const char * name,
                   sc_in<bool> & clk,
                   unsigned int periodMultiplier=1,
                   unsigned int startMultiplier=0);

scml_divided_clock(const char * name,
                   scml_clock_if & clk,
                   unsigned int periodMultiplier=1,
                   unsigned int startMultiplier=0);

scml_divided_clock(const char * name,
                   unsigned int periodMultiplier=1,
                   unsigned int startMultiplier=0);
```

where:

name	Specifies a name for the clock object.
clk	Specifies the clock from which this object is derived.
periodMultiplier	Specifies the factor by which the period is multiplied.
startMultiplier	Specifies the factor by which the start time is multiplied.

The default values are such that a clone of the incoming clock is obtained.

The following functions are provided to connect the divided clock to its input clock.

```
void bind(sc_in<bool> &);
void bind(scml_clock_if &);
void operator()(sc_in<bool> &);
void operator()(scml_clock_if &);
```

In addition to the API of an `scml_clock`, the following functions are available to set properties:

```
void set_divider(unsigned int div);
unsigned int get_divider() const;
```

Changes the clock period of a divided clock in multiples of the original clock period. The *original clock period* is the parent's clock period multiplied by the period multiplier constructor argument. For example, if the parent clock period is `p` and the divided clock period multiplier constructor argument is `2`, the original period is two times `p`.
`set_divider(4)` indicates that the new period is four times the original period, that is `8p`.

The majority of methods of an `scml_divided_clock` behaves the same as with an `scml_clock`, with the following exceptions:

```
bool is_master() const;
```

Always returns `false`.

```
bool disabled();
```

Tests whether the clock is disabled by the `disable()`/`enable()`.

```
bool running();
```

Tests whether the clock is running. This is the case if it is not disabled, and the master clock is running.

```
double get_duty_cycle() const;
void set_duty_cycle(double d);
```

Gets and sets the duty cycle of the divided clock. By default, a divided clock inherits the duty cycle from its parent clock. Any change of the duty cycle of the master clock will also become visible on the divided clock. This behavior changes after the duty cycle of the divided clock has been set explicitly by a call `set_duty_cycle()`. After such a call, the divided clock maintains its own value for the duty cycle and the value of the master clock is no longer used. The value provided to `set_duty_cycle()` must be larger than `0.0` and smaller than `1.0`.

The following functions to set properties are not supported by a divided clock and must not be called:

- `set_start_time(const sc_core::sc_time&)`
- `set_period(const sc_core::sc_time&)`
- `set_posedge_first(bool)`

3.3 Modeling Objects for Clocks (Clock Objects)

- scml_clock_gate
- scml_clock_counter

3.3.1 scml_clock_gate

`scml_clock_gate` is a module which takes a clock and an enable signal as input and produces a gated clock as output. If the enable signal is `true`, the output clock equals the input clock. If the enable signal is `false`, the output clock is disabled.

A clock gate has a clock and an enable input port:

```
sc_in<bool> clk;
sc_in<bool> en;
```

The following constructor is available:

```
scml_clock_gate(sc_module_name name);
```

where *`name`* specifies a name for the clock object.

This constructor is explicit.

The `scml_clock_gate` modeling object implements the interface `scml_clock_if`. It can, therefore, be used like an `sc_clock` or an `scml_clock` object. For an API reference, see “scml_clock” on page 68.

The majority of methods of an `scml_clock_gate` behaves the same as with an `scml_clock`, with the following exceptions:

```
bool is_master() const;
```

Always returns `false`.

```
bool disabled();
```

Tests whether the clock is disabled by the *`en`* input port.

```
bool running();
```

Tests whether the clock is running. This is the case if it is enabled by the *`en`* input port, and the master clock is running.

The following functions are not supported by a clock gate and must not be called:

- `enable()`
- `disable()`
- `set_duty_cycle(double)`
- `set_start_time(const sc_core::sc_time&)`
- `set_period(const sc_core::sc_time&)`
- `set_posedge_first(bool)`
- `set_period_multiplier(double)`

3.3.2 scml_clock_counter

An object of type `scml_clock_counter` needs to be attached to a clock. It is used to get the number of clock cycles that have happened in a certain period. The value of the counter is incremented at every clock cycle. Its initial value is `0`.

The following type definitions are available:

```
typedef sc_dt::uint64 data_type;
```

Objects of type `scml_clock_counter` can be constructed using one of the following constructors:

```
scml_clock_counter(const char * name,
                   scml_clock_if & clk);
scml_clock_counter(const char * name);
```

where:

name	Specifies a name for the clock object.
clk	Specifies the clock that should be used to determine this counter value.

The single-argument constructor is explicit.

The counter can be manipulated by means of the following sets of functions:

```
data_type get_count() const;
void set_count(data_type var);
```

Gets and sets the counter.

```
data_type read() const;
void write(const data_type var);
```

Read and write function.

```
operator const data_type() const;
data_type operator=(const data_type var);
```

Accesses the `scml_clock_counter` as a variable.

The following functions are provided to connect the clock counter to its input clock.

```
void bind(scml_clock_if &);
void operator()(scml_clock_if &);
```

3.4 Base Classes

- scml2::clocked_module

3.4.1 scml2::clocked_module

The class `clocked_module` is the lower level user API for SCML clock tick callbacks. The intended use is that a module can receive clock tick callbacks by inheriting from the `scml2::clocked_module` class and by implementing the virtual method `handle_clock_tick`.

A `clocked_module` has a fixed association with one SCML clock interface, it is not allowed to change the clock used by the `clocked_module` at runtime.

A clock tick callback can be requested by the method `request_clock_tick` callback. If there is already a callback request pending, the earlier one of the two will be maintained. It is possible to re-trigger or cancel a pending request.

Objects of type `scml2::clocked_module` can be constructed using one of the following constructors:

```
clocked_module(scml_clock_if* clock=0);
clocked_module(sc_core::sc_in<bool>& p);
```

where:

clock	Specifies the SCML clock object that is associated with the clocked module.
p	Specifies the input port through which the clocked module is bound to its associated SCML clock.

Clocked modules can be constructed and destructed at any time during a simulation run. If a clocked module is destructed and there is still a pending request for a clock tick callback for that object, it will automatically be canceled.

The following functions are provided to connect the clocked module to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_module` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_module` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

```
void request_clock_trigger(long long clock_ticks_to_skip);
```

Schedules a callback (that is, call to function `handle_clock_tick`) for a future clock tick. The argument `clock_ticks_to_skip` defines the number of ticks that shall be skipped from now. This function can be called at any SystemC time.

If called while a previous clock tick callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

If a clock trigger was requested, then the requested/scheduled clock tick can be retrieved by a call to method `get_scheduled_clock_tick()`.

For example, after a call of `request_clock_trigger(0)`, the clock tick count returned by `get_scheduled_clock_tick()` will be one larger than the current clock tick count of the associated SCML clock.

```
void request_clock_trigger(const sc_core::sc_time& delay);
```

Schedules a callback (that is, call to function `handle_clock_tick`) for a future clock tick after the time defined by the argument `delay`. The argument `delay` is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling, in order to synchronize to the next clock tick after the given local time argument.

If called while a previous clock tick callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled and the request is scheduled for the new earlier tick.

`virtual void handle_clock_tick()=0;`

When a requested clock tick callback expires, the SCML clock calls the method `handle_clock_tick()`. The call happens from within the context of an `SC_METHOD` that is owned by the clock object. This is an abstract method that shall be implemented by the user in the derived class in order to provide the functionality that shall be executed on a clock tick callback.

Caution You must not call `next_trigger()` from within a clock tick callback, since this will break the SCML clock mechanism and will result in a fatal misbehavior of the clock system.

`bool is_clock_trigger_requested() const;`

Returns `true` if a clock tick callback was requested and is still pending.

`void cancel_clock_trigger();`

Cancels a pending clock tick callback request. It does nothing if no clock tick callback request is currently pending. During the processing of a clock tick (that is within a `handle_clock_tick()` call), it is not permitted to cancel a clock tick callback request that was scheduled for the currently processed clock tick.

`unsigned long long get_scheduled_clock_tick() const;`

In case of a pending clock tick callback, this method returns the clock tick for which the tick callback was requested/scheduled. If no clock tick callback is pending, then the returned value is undefined.

3.5 Modeling Objects for Base Classes (Modeling Objects)

- scml2::clocked_timer

3.5.1 scml2::clocked_timer

This is a modeling object that provides a timer callback mechanism based on an SCML clock. The `clocked_timer` object is based on the `clocked_module` object and is similar to the `clocked_callback` object. The timer can be started to expire after a number of clock ticks. When it expires, it calls the callback that is registered with the object. The timer can be configured to expire once, multiple times or to run forever. It is possible to stop and resume the timer.

The following type definition is available:

`enum eState { eS_Idle=0, eS_Running, eS_Stopped };`

Objects of type `clocked_timer` can be constructed using the following constructor:

`clocked_timer(const std::string& *name*, scml_clock_if* *clock*=0);`

where:

name	Specifies a name for the clocked timer object.
clock	Specifies the SCML clock object that is associated with the clocked timer object.

Clocked timer objects can be constructed and destructed at any time during a simulation run

The following functions are provided to connect the clocked timer to an SCML clock.

`void set_clock(scml_clock_if* *clock*);`

Associates the `clocked_timer` with the provided SCML clock.

`void set_clock(sc_core::sc_in<bool>& p);`

Associates the `clocked_timer` with the SCML clock, which is bound to the provided input port.

`scml_clock_if* get_clock() const;`

Returns the associated SCML clock.

Registering a callback for alarm notifications is typically done using the following macro:

`m_clk_timer.set_callback(SCML2_CLOCKED_CALLBACK(clock_cb));`

`void start(long long ticks_per_period, long long shot_count=-1);`

Starts the timer. It will expire after `ticks_per_period` ticks. For example, a timer started with `start(1)` will expire at the next clock tick. With `ticks_per_period=0`, the timer does not start.

The argument `shot_count` specifies how many alarm callbacks will be triggered by the timer with a distance of `ticks_per_period` ticks. With `shot_count=1`, the timer expires once. With `shot_count=-1` (default), the timer runs forever or until it is stopped by calling `stop()`. With `shot_count=0` the timer does not start.

If `start()` is called while the timer is running, it will first be stopped before it is restarted. If a running timer is started with `ticks_per_period=0` or `shot_count=0`, the timer is stopped and left in the idle state, so that it is not possible to resume it at a later time.

`void start(const sc_core::sc_time& period, long long shot_count=-1);`

Starts the timer. This is a convenience method comparable to `start(long long, long long)`. The period in clock ticks is derived by the `sc_time` argument period, by rounding up to the next higher multiple of the current clock period. For more details, see the description of the method, `start(long long, long long)`

`void stop();`

Stops a running timer. The timer is left in a state that allows it to be resumed at a later time. It has no effect, if called on an idle or already stopped timer.

`bool resume_stopped();`

Resume a stopped timer. Only timers that have been stopped by `stop()` can be resumed. In that case, the timer continues to run with the status at which it was stopped. If the timer was stopped in the middle of a period, than the timer finishes the already started period. For example, a timer started at clock tick `100` with `period=10` and `shot_count=3`, stopped at time 115 (after one alarm and another half period), and resumed at time 200, will expire at the times 205 and 215.

It returns `true` on success. `False` is returned, if it is called on an idle or already running timer.

`unsigned long long get_counter_value();`

Returns the value of the internal counter of the timer. It counts the number of clock ticks since the last start or restart.

`eState get_state();`

Returns the state of the timer. In order to avoid dependency on scheduling order, the state at the beginning of the current SystemC time is returned. A change to the state will only become visible after the SystemC time advances. If called from the timer callback, it returns the state before the current clock cycle.

`bool is_active();`

Returns `true`, if the timer is running. In order to avoid dependency on scheduling order, the activity state at the beginning of the current SystemC time is returned. A change to the state will only become visible after the SystemC time advances. If called from the timer callback, it returns the state before the current clock cycle.

`long long get_remaining_shot_count();`

Returns the number of remaining shots. In order to avoid dependency on scheduling order, the activity state at the beginning of the current SystemC time is returned. A change to the state will only become visible after the SystemC time advances. If called from the timer callback, it returns the value before the current clock cycle.

3.6 Convenience Classes

- scml2::clocked_callback
- scml2::clocked_event

3.6.1 scml2::clocked_callback

The class `scml2::clocked_callback` is a convenience class that forwards a clock tick callback to any member function of a module without the need to inherit from the `clocked_module` base class. This makes it easier to have multiple clock callbacks within a single module. Since the callback is received via a `clocked_callback`, it involves an internal method redirection which is slightly slower than a callback received via inheriting the module directly from `clocked_module`.

Objects of type `clocked_callback` can be constructed using the following constructor:

`clocked_callback(const std::string& *name*, scml_clock_if* *clock*=0)`

where:

name	Specifies a name for the clocked callback object.
clock	Specifies the SCML clock object that is associated with the clocked callback object.

Clocked callbacks can be constructed and destructed at any time during a simulation run. If a clocked callback is destructed and there is still a pending callback request for that object, it will automatically be canceled.

The following functions are provided to connect the clocked callback to an SCML clock.

`void set_clock(scml_clock_if* *clock*);`

Associates the `clocked_callback` with the provided SCML clock.

`void set_clock(sc_core::sc_in<bool>& *p*);`

Associates the `clocked_callback` with the SCML clock, which is bound to the provided input port.

`scml_clock_if* get_clock() const;`

Returns the associated SCML clock.

Registering a method to be called as clock callback is typically done using the following macro:

```
m_clk_cb.set_callback(SCML2_CLOCKED_CALLBACK(clock_cb));
```

`void request_trigger(long long clock_ticks_to_skip);`

Schedules a callback for a future clock tick. The argument `clock_ticks_to_skip` defines the number of ticks that shall be skipped from now. This function can be called at any SystemC time.

If called while a previous callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

If a callback was requested, then the requested/scheduled clock tick can be retrieved by a call to method `get_scheduled_clock_tick()`.

For example, after a call of `request_trigger(0)`, the clock tick count returned by `get_scheduled_clock_tick()` will be one larger than the current clock tick count of the associated SCML clock.

```
void request_trigger(const sc_core::sc_time& delay);
```

Schedules a callback for a future clock tick after the time defined by the argument `delay`. The argument `delay` is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling in order to synchronize to the next clock tick after the given local time argument.

If called while a previous callback request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

```
bool is_trigger_requested() const;
```

Returns `true` if a callback was requested and is still pending.

```
void cancel_trigger();
```

Cancels a pending callback request. It does nothing, if no callback request is currently pending. During the processing of a callback, it is not permitted to cancel a callback request that was scheduled for the currently processed clock tick.

In certain cases, it can also be useful to create a one-shot callback. Here, the idea is to have a class method to be called once on a clock tick. For this purpose, a set of global functions are provided that create a `clocked_callback` and trigger a request after a certain number of ticks or a certain SystemC delay. This can be used for example, to trigger a certain behavior after a register access.

```
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
    scml_clock_if* clock, long long clock_ticks_to_skip,
    MOD_TYPE* mod, FUNCPTR_TYPE func
);
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
    sc_core::sc_in<bool>& clock_port, long long clock_ticks_to_skip,
    MOD_TYPE* mod, FUNCPTR_TYPE func
);
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
    scml_clock_if* clock, const sc_core::sc_time& delay,
    MOD_TYPE* mod, FUNCPTR_TYPE func
);
template <class MOD_TYPE, typename FUNCPTR_TYPE>
void request_clocked_method_callback(
    sc_core::sc_in<bool>& clock_port, const sc_core::sc_time& delay,
    MOD_TYPE* mod, FUNCPTR_TYPE func
```

3.6.2 scml2::clocked_event

An `scml2::clocked_event` is a convenience class that allows a SystemC method or thread to wait until a certain clock tick happens. The `clocked_event` object has the same set of APIs as a `clocked_module`, but also provides a `wait()` API, so that a thread can wait for a number of clock cycles or for the first clock tick after a certain delay. It has a `wait_for_trigger()` API that halts execution, until a certain clock tick specified by `requested_trigger` (similar to the `clocked_module`) happens. This object also has a `next_trigger()` API to mimic dynamic sensitivity of the methods. In this case, the trigger will be aligned with clock edges.

Objects of type `clocked_event` can be constructed using one of the following constructors:

```
clocked_event(const std::string& name, scml_clock_if* clock=0);
clocked_event(const std::string& name, sc_core::sc_in<bool>& p);
```

where:

name	Specifies a name for the clocked event object.
clock	Specifies the SCML clock object that is associated with the clocked event object.
p	Specifies the input port through which the clocked event is bound to its associated SCML clock.

Clocked events can be constructed and destructed at any time during a simulation run. If a clocked callback is destructed and there is still a trigger pending, it will automatically be canceled. If a thread or a method is waiting for a clock tick, it will never wake up or be activated again.

The following functions are provided to connect the clocked event to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_event` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_event` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

```
void request_trigger(long long clock_ticks_to_skip);
```

Schedules a trigger for a future clock tick, which will wake up a SystemC thread (which is called `wait()`), or which re-activates a SystemC methods (which is called `next_trigger()`). The argument `clock_ticks_to_skip` defines the number of ticks that shall be skipped from now. It can be called at any SystemC time.

If called while a previous trigger request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

If a trigger was requested, then the requested/scheduled clock tick can be retrieved by a call to method `get_scheduled_clock_tick()`.

For example, after a call of `request_clock_trigger(0)`, the clock tick count returned by `get_scheduled_clock_tick()` will be one larger than the current clock tick count of the associated SCML clock.

```
void request_trigger(sc_core::sc_time delay);
```

Schedules a trigger for a future clock tick after the time defined by the argument `delay`. The argument `delay` is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling, in order to synchronize to the next clock tick after the given local time argument.

If called while a previous trigger request is still pending, the earlier one of the two will be maintained. That is, the new request is ignored if it targets the same or a later clock tick. If it targets an earlier clock tick, then the previous request is canceled, and the request is scheduled for the new earlier tick.

`bool is_trigger_requested() const;`

Returns `true` if a trigger was requested and is still pending.

`void cancel_trigger();`

Cancels a pending trigger request. It does nothing if no trigger is currently pending. During the processing of a clock tick (that is from a callback, or a `handle_clock_tick()` call), it is not permitted to cancel a trigger request that was scheduled for the currently processed clock tick.

`void waitfor_trigger();`

Blocks the calling SystemC thread, until a clocked trigger happens. The clocked trigger is requested by a call to the method `request_trigger(ticks/delay)`. The blocked thread wakes up when the requested clock tick happens. It must only be called from the context of a SystemC thread.

`void wait(long long clock_ticks_to_skip);`

Blocks the calling SystemC thread, until the specified clock tick happened. The clock tick is specified by the argument `clock_ticks_to_skip`, which defines the number of ticks that shall be skipped from now. It can be called at any SystemC time, but only from the context of a SystemC thread.

It is possible to wake up the thread at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the thread at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

`void wait(sc_core::sc_time delay);`

Blocks the calling SystemC thread, until the specified clock tick happened. The clock tick is specified by the argument `delay`, which is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling in order to synchronize to the next clock tick after the given local time argument. It can be called at any SystemC time, but only from the context of a SystemC thread.

It is possible to wake up the thread at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the thread at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

`void next_trigger();`

Temporarily overrides the static sensitivity list of the calling SystemC method. The method will be reactivated when a certain clock tick happens. With this version of the `next_trigger()` function, the triggering clock tick is left undefined. It must be scheduled by calling `request_trigger(ticks/delay)`.

`void next_trigger(long long clock_ticks_to_skip);`

Temporary overrides the static sensitivity list of the calling SystemC method. The method will be reactivated when the specified clock tick happens. The clock tick is specified by the argument `clock_ticks_to_skip`, which defines the number of ticks that shall be skipped from now. It can be called at any SystemC time, but only from the context of a SystemC method.

It is possible to reactivate the method at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the method at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

```
void next_trigger(sc_core::sc_time delay);
```

Temporary overrides the static sensitivity list of the calling SystemC method. The method will be reactivated when the specified clock tick happens. The clock tick is specified by the argument `delay`, which is a SystemC time that is interpreted relative to the current SystemC time `sc_time_stamp()`. This method will usually be used by TLM2 models with temporal decoupling in order to synchronize to the next clock tick after the given local time argument. It can be called at any SystemC time, but only from the context of a SystemC method.

It is possible to reactivate the method at an earlier time by calling `request_trigger(ticks/delay)` with an argument that specifies an earlier clock tick. In order to wake up the method at a later time, it is first necessary to cancel the pending trigger with a call to `cancel_trigger()`.

3.7 Modeling Objects for Convenience Classes (Convenience Objects)

- scml2::clocked_peq_container
- scml2::clocked_peq

3.7.1 scml2::clocked_peq_container

A typical problem in TLM2 models using the non-blocking APIs is that it can be required to manage multiple outstanding transactions, possibly coming with different timing annotations from different initiators. The `clocked_peq_container` is a convenience object to help in this case. It can be used to buffer payloads arriving in the model. Payloads are pushed into the container tagged with an arrival stamp (clock tick counts to be precise). The container behaves like a list of payload that is sorted by the arrival stamp. There is an API to iterate over the payloads in the buffer, the iterator will only provide with those payloads for which the arrival time is in the past (compared to the current SystemC time when iterating). In this way, this object helps to comply with the basic clocked modeling rules. Because the timestamp is stored based on tick counts, it is possible to adjust arrival times when clock parameters change (period/enable/disable).

The API of the container conforms to the STL and provides iteration into the forward direction of the list (from old to new payloads). There is a constant and a non-constant version of the iterator. The constant iterator provides a faster iteration mechanism. The non-constant iterator allows removing the payload it is pointing to from the list.

The `scml2::clocked_peq_container` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_container;
```

The following type definitions for iterators are available:

```
typedef clocked_peq_container_iterator<PAYLOAD> iterator;
typedef clocked_peq_container_const_iterator<PAYLOAD> const_iterator;
```

Objects of type `clocked_peq_container` can be constructed using one of the following constructors:

```
clocked_peq_container(scml_clock_if* clock=0);
clocked_peq_container(sc_core::sc_in<bool>& p);
```

where:

clock	Specifies the SCML clock object that is associated with the clocked PEQ container object.

p	Specifies the input port through which the clocked PEQ container is bound to its associated SCML clock.

The following functions are provided to connect the clocked PEQ container to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_peq_container` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_peq_container` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

```
bool notify(PAYLOAD* payload, const sc_core::sc_time& arrival_local_time);
```

Pushes a payload into the container that was received by a temporal decoupled transport at local time `arrival_local_time`. The local time argument is given as SystemC time. It is converted to the clock tick count by rounding down to the last clock tick.

```
bool notify(
    PAYLOAD* payload, const sc_core::sc_time& arrival_local_time,
    long long delay_ticks
);
```

Pushes a payload into the container that was received by a temporal decoupled transport. The local time is defined by two arguments, the SystemC time `arrival_local_time` and the additional number of clock ticks `delay_ticks`. The internal arrival clock tick count is calculated by first converting `arrival_local_time` to the clock tick count by rounding down to the last clock tick, and then incrementing it by `delay_ticks`.

```
bool notify(void* payload, unsigned long long clock_ticks_to_skip);
```

Pushes a payload into the container that was received by a temporal decoupled transport at the local time that is `clock_ticks_to_skip` clock ticks in the future.

```
bool notifyAt(void* payload, unsigned long long arrival_tick_count);
```

Pushes a payload into the container that was received by a temporal decoupled transport at the local time represented by the clock tick count `arrival_tick_count`.

```
bool is_empty() const;
```

Returns `true` if the container is empty. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

```
const_iterator begin() const;
```

Returns a constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

```
iterator begin();
```

Returns a non-constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

```
const_iterator end() const;
```

Returns a constant iterator that points to the end of the internal sorted list. This only considers visible payload `(timestamp` < `get_clock()->get_tick_count()).`

```
iterator end();
```

Returns a non-constant iterator that points to the end of the internal sorted list. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`PAYLOAD* get_next();`

Extracts the first element from the internal sorted list. It returns a null-pointer if the list is empty. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`void remove(const iterator& pos);`

Removes the payload at the position, which is given by the iterator argument `pos`. The payload itself is not deleted.

This operation invalidates all the iterators, which are currently pointing into the container. Managing the iterators is the user's responsibility.

`bool remove(PAYLOAD* payload);`

Removes the payload given by the `payload` argument. The payload itself is not deleted. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`). It returns `true` if the payload was found and removed. `False` is returned if the payload was not found. If the same payload is contained multiple times, than only the first occurrence is removed.

If this operation succeeds, it invalidates all iterators which are currently pointing into the container. Managing the iterators is the user's responsibility.

`bool has_more_events() const;`

Returns `true` if more payload is available in the container. In contrast to the method `is_empty()`, it considers not only the visible, but all payload. This method is usually used, if all visible/past payload has been processed/removed, in order to check, if more processing has to be done at a future time.

`unsigned long long get_next_event_arrival_tick() const;};`

Returns the arrival clock tick of the next payload in the sorted list of the container. This method must not be called if `has_more_events()` returned `false`.

Note This method does not only consider the visible, but all payload. This method is usually used when all current processing has been done, in order to retrieve the time for which more processing shall be scheduled.

An object of class `clocked_peq_container_iterator` is a pointer into a `clocked_peq_container` that allows iterating in forward direction over the sorted list of the container. This non-constant version of iterators can be used to remove that payload from the container, which it is currently pointing to. Iterating is slightly slower than with the constant version `clocked_peq_container_const_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a clocked PEQ container.

The `scml2::clocked_peq_container_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_container_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

`PAYLOAD* operator*() const;`

Returns the payload pointed to by the iterator.

`PAYLOAD* operator->() const;`

Returns the payload pointed to by the iterator.

```
self& operator++();
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the new position.

```
self operator++(int);
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the old position.

```
bool operator==(const self& x) const;
```

Compares the iterator to the other iterator `x`. It returns `true`, if both iterators are pointing to the same position.

```
bool operator!=(const self& x) const;
```

Compares the iterator to the other iterator `x`. It returns `true`, if both iterators are pointing to different positions.

An object of class `clocked_peq_container_const_iterator` is a pointer into a `clocked_peq_container` that allows iterating in forward direction over the sorted list of the container. This constant iterator cannot be used to remove that payload from the container, which it is currently pointing to. Iterating is slightly faster than with the non-constant version `clocked_peq_container_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a container.

The `scml2::clocked_peq_container_const_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_container_const_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

```
const PAYLOAD* operator*() const;
```

Returns the payload pointed to by the iterator.

```
const PAYLOAD* operator->() const;
```

Returns the payload pointed to by the iterator.

```
self& operator++();
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the new position.

```
self operator++(int);
```

Moves the iterator forward to the next entry in the container. It returns an iterator, that points to the old position.

```
bool operator==(const self& x) const;
```

Compares the iterator to the other iterator `x`. It returns `true`, if both iterators are pointing to the same position.

```
bool operator!=(const self& x) const;
```

Compares the iterator to the other iterator `x`. It returns `true`, if both iterators are pointing to different positions.

3.7.2 scml2::clocked_peq

This is a modeling object similar to the `clocked_peq_container`. It has a set of additional features:

- It allows registering a callback, which will be triggered whenever an element from the payload buffer becomes available.
- The callback will be called when the SystemC time has moved forward to the point of the arrival/notified time, that got stored with the payload in the buffer.
- It is possible to block the callback triggers. When blocked, the callback will not be triggered, until it is unblocked.
- When the callback is unblocked and the buffer contains a payload with arrival time that is in the past, the callback will be triggered at the next clock edge following the unblock time.

The `scml2::clocked_peq` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq;
```

The following type definitions for iterators are available:

```
typedef clocked_peq_iterator<PAYLOAD> iterator;
typedef clocked_peq_const_iterator<PAYLOAD> const_iterator;
```

Objects of type `clocked_ peq` can be constructed using one of the following constructors:

```
clocked_peq(scml_clock_if* clock=0);
clocked_peq(sc_core::sc_in<bool>& p);
```

where:

clock	Specifies the SCML clock object that is associated with the clocked PEQ container object.
p	Specifies the input port through which the clocked PEQ container is bound to its associated SCML clock.

The following functions are provided to connect the clocked PEQ to an SCML clock.

```
void set_clock(scml_clock_if* clock);
```

Associates the `clocked_peq` with the provided SCML clock.

```
void set_clock(sc_core::sc_in<bool>& p);
```

Associates the `clocked_peq` with the SCML clock, which is bound to the provided input port.

```
scml_clock_if* get_clock() const;
```

Returns the associated SCML clock.

Registering a callback for notifications of new payload is typically done using the following macro:

```
m_clk_peq.set_callback(SCML2_CLOCKED_CALLBACK(clock_cb));
```

```
bool notify(PAYLOAD* payload, const sc_core::sc_time& arrival_local_time);
```

Pushes a payload into the PEQ that was received by a temporal decoupled transport at local time `arrival_local_time`. The local time argument is given as SystemC time. It is converted to the clock tick count by rounding down to the last clock tick.

```
bool notify(PAYLOAD* payload, const sc_core::sc_time& arrival_local_time, long long
delay_ticks);
```

Pushes a payload into the PEQ that was received by a temporal decoupled transport. The local time is defined by two arguments, the SystemC time `arrival_local_time` and the additional number of clock ticks `delay_ticks`. The internal arrival clock tick count is calculated by first converting `arrival_local_time` to the clock tick count by rounding down to the last clock tick, and then incrementing it by `delay_ticks`.

`bool notify(void* payload, unsigned long long clock_ticks_to_skip);`

Pushes a payload into the PEQ that was received by a temporal decoupled transport at the local time that is, `clock_ticks_to_skip` clock ticks in the future.

`bool notifyAt(void* payload, unsigned long long arrival_tick_count);`

Pushes a payload into the PEQ that was received by a temporal decoupled transport at the local time represented by the clock tick count `arrival_tick_count`.

`bool is_empty() const;`

Returns `true` if the PEQ is empty. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`const_iterator begin() const;`

Returns a constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`iterator begin();`

Returns a non-constant iterator that points to the beginning of the internal sorted list. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`const_iterator end() const;`

Returns a constant iterator that points to the end of the internal sorted list. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`iterator end();`

Returns a non-constant iterator that points to the end of the internal sorted list. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`PAYLOAD* get_next();`

Extracts the first element from the internal sorted list. It returns a null pointer if the list was empty. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`).

`void remove(const iterator& pos);`

Removes the payload at position which is given by the iterator argument `pos`. The payload itself is not deleted.

This operation invalidates all the iterators which are currently pointing into the container. Managing the iterators is the user's responsibility.

`bool remove(PAYLOAD* payload);`

Removes the payload given by the ***`payload`*** argument. The payload itself is not deleted. This only considers visible payload (`timestamp` < `get_clock()->get_tick_count()`). It returns `true` if the payload was found and removed. `False` is returned if the payload was not found. If the same payload is contained multiple times, than only the first occurrence is removed.

If this operation succeeds, it invalidates all those iterators, which are currently pointing into the container. Managing the iterators is the user's responsibility.

`bool has_more_events() const;`

Returns `true` if more payload is available in the container. In contrast to the method `is_empty()`, this considers not only the visible, but all payload. This method is usually used, if all visible/past payload has been processed/removed. In order to learn, if more processing has to be done at a future time, even if no new payload is pushed into the container.; an unblocked PEQ schedules another callback notification for the future.

`unsigned long long get_next_event_arrival_tick() const;`

Returns the arrival clock tick of the next payload in the PEQ. This is the time tick at which an unblocked PEQ will send the next callback. This method must not be called if `has_more_events()` returned `false`!

Note This method does not only consider the visible, but all payload. This method is usually used when all current processing has been done, in order to retrieve the time, when the next callback will notify to continue with more processing.

`void block();`

Blocks the sending of callbacks for the next payload arrival time.

`void unblock(const sc_core::sc_time& local_time_before_clock_tick);`

Schedules to unblock the PEQ after a SystemC time interval of `local_time_before_clock_tick`. If at the time of unblocking, the PEQ contains a payload with an arrival time that lays in the past, then a callback is sent at the next clock edge following the unblock time.

For example: Calling `unblock(SC_ZERO_TIME)` on a blocked PEQ, that contains a visible payload (arrival time in the past) will schedule a callback for the next clock tick.

`void unblock(const sc_core::sc_time& local_time_before_clock_tick, long long delay_ticks);`

Schedules to unblock the PEQ after a SystemC time interval of `local_time_before_clock_tick` plus `delay_ticks` clock ticks. For more details, see the method `unblock(const sc_core::sc_time&)`.

`void unblock(unsigned long long clock_ticks_to_skip);`

Schedules to unblock the PEQ after `clock_ticks_to_skip` clock ticks. For more details, see the method `unblock(const sc_core::sc_time&)`.

For example: Calling `unblock(0)` on a blocked PEQ that contains a visible payload (arrival time in the past) will schedule a callback for the next clock tick.

`void unblockAt(unsigned long long tick_count_for_unblock);`

Schedules to unblock the PEQ for the time given by the clock tick count `tick_count_for_unblock`. If `tick_count_for_unblock` lays in the past, then the PEQ is unblocked at the next tick count. For more details, see the method `unblock(const sc_core::sc_time&)`.

For example: Calling `unblockAt(get_clock()->get_tick_count())` on a blocked PEQ that contains a visible payload (arrival time in the past) will schedule a callback for the next clock tick.

An object of class `clocked_peq_iterator` is a pointer into a `clocked_peq` that allows iterating in forward direction over the sorted list of the PEQ. This non-constant version of iterators can be used to remove the payload it is currently pointing to from the PEQ. Iterating is slightly slower than with the constant version `clocked_peq_const_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a clocked PEQ.

The `scml2::clocked_peq_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

`PAYLOAD* operator*() const;`

Returns the payload pointed to by the iterator.

`PAYLOAD* operator->() const;`

Returns the payload pointed to by the iterator.

`self& operator++();`

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the new position.

`self operator++(int);`

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the old position.

`bool operator==(const self& x) const;`

Compares the iterator to the other iterator `x`. It returns `true` if both iterators are pointing to the same position.

`bool operator!=(const self& x) const;`

Compares the iterator to the other iterator x. It returns `true` if both iterators are pointing to different positions.

An object of class `clocked_peq_const_iterator` is a pointer into a `clocked_peq` that allows iterating in forward direction over the sorted list of the PEQ. This constant iterator cannot be used to remove the payload it is currently pointing to from the PEQ. Iterating is slightly faster than with the non-constant version `clocked_peq_iterator`. Iterators are usually generated by the `begin()` and `end()` method of a container.

The `scml2::clocked_peq_const_iterator` class is templated with the underlying type of the payload:

```
template <typename PAYLOAD> class clocked_peq_const_iterator;
```

The API of the iterator conforms to the STL and provides the following methods:

`const PAYLOAD* operator*() const;`

Returns the payload pointed to by the iterator.

`const PAYLOAD* operator->() const;`

Return the payload pointed to by the iterator.

`self& operator++();`

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the new position.

`self operator++(int);`

Moves the pointer forward to the next entry in the PEQ. It returns an iterator that points to the old position.

`bool operator==(const self& x) const;`

Compares the iterator to the other iterator `x`. It returns `true` if both iterators are pointing to the same position.

```
bool operator!=(const self& x) const;
```

Compares the iterator to the other iterator `x`. It returns `true` if both iterators are pointing to different positions.

3.8 Code Example

This section describes how a programmable clock peripheral can be implemented based on `scml_clock`.

- Programmable Clock Peripherals

3.8.1 Programmable Clock Peripherals

Programmable clock peripherals and timers can be coded easily by using clock tick callbacks. This is achieved by deriving the Timer from `scml2::clocked_module` and implementing `handle_clock_tick()`:

```
SC_MODULE Timer : private scml2::clocked_module {
public:
  sc_in<bool> clk_in;
  [...]
private:
  [...]
  virtual void end_of_elaboration() {
    set_clock(clk_in);
  }
  virtual void handle_clock_tick();
};
```

The functionality of the peripheral is implemented in the callback function attached to a memory-mapped register, modeled as an object of type `scml_memory`.

For example, consider a module with the following data members:

```
scml_memory <unsigned int> CURRENT_VALUE_REG;
scml_memory <unsigned int> END_VALUE_REG;
unsigned long long mTimerStartTickCount;
```

Two call-back functions are registered with the `END_VALUE_REG` memory-mapped register:

```
MEMORY_REGISTER_READ(CURRENT_VALUE_REG, f_read_curr_value);
MEMORY_REGISTER_WRITE(END_VALUE_REG, f_write_end_value);
```

The implementation of the write callback writes a new value to `END_VALUE_REG` memory-mapped register and restarts the timer:

```
void f_write_end_value(unsigned int new_value, unsigned int, unsigned int) {
  END_VALUE_REG=new_value;
  cancel_clock_trigger();
  mTimerStartTickCount=get_clock()->get_tick_count();
  request_clock_trigger(END_VALUE_REG - 1);
}
```

The actual counter value of the timer is only calculated on demand. The implementation of the read callback takes the current tick count of the driving clock and subtracts the tick count from when the timer started:

```
unsigned int f_read_curr_value(unsigned int, unsigned int) {
  return (unsigned int) (get_clock()->get_tick_count() - mTimerStartTickCount);
}
```

Chapter 4 Modeling Utilities

This chapter describes:

- Port Utilities
- Parameters
- Reporting
- FastTrack

4.1 Port Utilities

This section describes:

- dmi_handler
- initiator_socket
- Pin Callback Functions

4.1.1 dmi_handler

`scml2::dmi_handler` is a convenience object that takes care of the DMI pointer requests and book keeping. The object is targeting protocols for which the DMI handling is the same as for the TLM2 base protocol. It may be required to create a dedicated DMI handler for other protocol definitions, for example, when address decoding can be influenced by other attributes like security and so on. The DMI handler is initialized by giving it an interface that it should use to initiate TLM2 transport calls. The model itself should then use the read and write APIs of the DMI handler to start a transaction. The DMI handler will first check if it does not have a pointer available to provide the read or write data, if not it will try to request a DMI pointer and if that fails it will forward the transaction of the TLM2 interface. The DMI handler also has APIs to control whether DMI accesses will be attempted.

The include file of the `dmi_handler` objects is `scml2/dmi_handler.h`.

The following sections describe:

- Methods that are available to configure the `dmi_handler` object:
- The access methods of the `dmi_handler` object.

The following methods are available to configure the `dmi_handler` object:

`void set_interface(tlm::tlm_fw_direct_mem_if<tlm::tlm_generic_payload>* `*`ifs`*`)`

Sets the forward DMI. This interface is used to request the DMI pointers.

`bool is_dmi_enabled()`

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

The `dmi_handler` object has the following access methods:

```
bool read(unsigned long long address,
          unsigned char* data,
          unsigned int dataLength,
          const unsigned char* byteEnables,
          unsigned int byteEnableLength,
          sc_core::sc_time& t)
bool write(unsigned long long address,
          const unsigned char* data,
          unsigned int dataLength,
          const unsigned char* byteEnables,
          unsigned int byteEnableLength,
          sc_core::sc_time& t)
bool read(unsigned long long address,
          unsigned char* data,
          unsigned int dataLength,
          sc_core::sc_time& t)
bool write(unsigned long long address,
          const unsigned char* data,
          unsigned int dataLength,
          sc_core::sc_time& t)
```

Try to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned. The *t* argument is incremented with the read or write latency, respectively.

```
bool read_debug(unsigned long long address,
                unsigned char* data,
                unsigned int dataLength)
                bool write_debug(unsigned long long address,
                const unsigned char* data,
                unsigned int dataLength)
```

Tries to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned.

```
bool transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
bool transport_debug(tlm::tlm_generic_payload& trans)
```

Try to do a DMI access. If a DMI access is not possible or if the access does not fit into one DMI range, `false` is returned. Otherwise the data is copied and `true` is returned.

```
void invalidate_direct_mem_ptr(sc_dt::uint64 startRange,
                               sc_dt::uint64 endRange)
```

Must be called when the DMI pointers have to be invalidated.

4.1.2 initiator_socket

`scml2::initiator_socket` is a convenience TLM2 initiator socket specifically targeting the LT coding style. It contains a `dmi_handler` to manage DMI accesses and it implements the `mappable_if` so that this socket can be used as a destinations for a router. The `initiator_socket` also gets a reference to a `quantum_keeper` that it will use to maintain timing annotation whenever the `dmi_handler` does an access of the forward transport interface of the socket. Transactions are initiated in the same way as for the `dmi_handler`; that is, via read and write APIs.

The `initiator_socket` object implements the mappable_if object, which means that it can be the destination for a mapped range of a router object.

The include file of the `initiator_socket` objects is `scml2/initiator_socket.h`.

The `initiator_socket` class is templated with the `BUSWIDTH`:

```
template <unsigned int BUSWIDTH> class initiator_socket
```

The following methods are available to configure the `initiator_socket` object:

```
template <typename T>
void set_quantumkeeper(T& quantumKeeper)
```

Sets the quantum keeper the socket should use. The registered class must implement the following methods (see the section on `tlm_quantumkeeper` in the *IEEE Std 1666 TLM-2.0 Language Reference Manual*):

```
void inc(const sc_core::sc_time& t)
void set(const sc_core::sc_time& t)
bool need_sync() const
void sync()
sc_core::sc_time get_local_time() const
```

If a quantum keeper is set, the socket will pass the local time when doing a bus access and increment the local time when the timing annotation was incremented by the DMI access or bus access. If needed (`need_sync()` returns `true`), the socket will synchronize the quantum keeper after incrementing the local time.

If no quantum keeper is set, `sc_core::SC_ZERO_TIME` will be passed and `wait()` will be called if the timing annotation was incremented.

```
void set_endianness(tlm::tlm_endianness endianness)
```

Sets the endianness of the initiator mode. If the endianness is different from the host endianness, the socket converts the address and data before doing the access.

```
bool is_dmi_enabled()
```

Returns `true` if DMI accesses are allowed for the object, `false` otherwise. DMI is enabled by default.

```
void enable_dmi()
void disable_dmi()
```

Enables/disables DMI accesses for the object.

The following access methods are available:

```
template <typename DT>
bool read(unsigned long long address, DT& data)
template <typename DT>
bool write(unsigned long long address, const DT& data)
```

Access methods to do single-word or subword accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned, otherwise `true` is returned. If a quantum keeper is set, the local time is passed with the bus access and the local time of the quantum keeper is incremented with the returned timing annotation. If no quantum keeper is set, `SC_ZERO_TIME` is passed and `wait()` is called if the timing annotation was incremented.

```
template <typename DT>
bool read(unsigned long long address, DT* data, unsigned int count)
template <typename DT>
bool write(unsigned long long address, const DT* data, unsigned int count)
```

Access methods for burst accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned; otherwise `true` is returned. If a quantum keeper is set, the local time is passed with the bus access and the local time of the quantum keeper is incremented with the returned timing annotation. If no quantum keeper is set, `SC_ZERO_TIME` is passed and `wait()` is called if the timing annotation was incremented.

```
template <typename DT>
bool read(unsigned long long address, DT& data, sc_core::sc_time& t)
template <typename DT>
bool write(unsigned long long address, const DT& data, sc_core::sc_time& t)
```

Access methods to do single-word or subword accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned, otherwise `true` is returned.
The time argument is passed with the `b_transport()` call. If a quantum keeper was set in the socket, it will be ignored.

```
template <typename DT>
bool read(unsigned long long address, DT* data, unsigned int count,
                                               sc_core::sc_time& t)
template <typename DT>
bool write(unsigned long long address, const DT* data, unsigned int count)
```

Access methods for burst accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If this access fails with an error response, `false` is returned; otherwise `true` is returned. The time argument is passed with the `b_transport()` call. If a quantum keeper was set in the socket, it will be ignored.

```
template <typename DT>
bool read_debug(unsigned long long address, DT& data)
template <typename DT>
bool write_debug(unsigned long long address, const DT& data)
```

Access methods to do single-word or subword debug accesses. The data passed must be in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If the debug bus access did not succeed, `false` is returned; otherwise `true` is returned.

```
template <typename DT>
bool read_debug(unsigned long long address, DT* data, unsigned int count)
template <typename DT>
bool write_debug(unsigned long long address, const DT* data, unsigned int count)
```

Access methods to do burst debug accesses. The passed data pointer should contain an array of words in arithmetic format (host endianness). If the endianness of the socket is different from the host endianness, the address and data are converted before doing the access. First a DMI access is done. If this fails, a bus access (`b_transport()`) is done. If the debug bus access did not succeed, `false` is returned; otherwise `true` is returned.

```
void b_transport(tlm::tlm_generic_payload& trans, sc_core::sc_time& t)
unsigned int transport_dbg(tlm::tlm_generic_payload& trans)
tlm::tlm_sync_enum nb_transport_fw(tlm::tlm_generic_payload& trans,
                         tlm::tlm_phase& phase, sc_core::sc_time& t)
bool get_direct_mem_ptr(tlm::tlm_generic_payload& trans, tlm::tlm_dmi& dmiData)
```

TLM2 access methods. First a DMI access is tried. If this fails, a bus access is done. No endianness conversions are done; the passed transaction should already be in the correct format.

The following methods are available to register or unregister a backward path interface to the `initiator_socket`. For more information, see the *Accellera IEEE 1666 LRM Language Reference Manual*.

```
void register_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface)
void unregister_bw_direct_mem_if(tlm::tlm_bw_direct_mem_if* bwInterface)
```

Register or unregister `tlm::tlm_bw_direct_mem_if` to the `initiator_socket`. The `invalidate_direct_mem_ptr` method of all registered interfaces will be called in case the `invalidate_direct_mem_ptr` call is done on the backward path of the `initiator_sockets`. Multiple interfaces can be registered. In such cases, the call will be forwarded to all registered interfaces.

```
typedef tlm::tlm_bw_nonblocking_transport_if<tlm::tlm_generic_payload, tlm::tlm_phase>
                                                                BwTransportIf
virtual void register_bw_transport_if(BwTransportIf* bwInterface)
virtual void unregister_bw_transport_if(BwTransportIf* bwInterface)
```

Register or unregister `tlm::tlm_bw_nonblocking_transport_if` to the `initiator_socket`. The `nb_transport_bw` method of the registered interface will be called in case the `nb_transport_bw` call is done on the backward path of the `initiator_sockets`. Only one `tlm::tlm_bw_nonblocking_transport_if` can be registered to the `initiator_socket`.

4.1.3 Pin Callback Functions

The following functions are available for registering user callbacks on changes of input pins:

- The convenience functions for registering user callbacks on pins:

```
set_change_callback(pin, object, callback);
set_change_callback(pin, object, callback, tag);
set_posedge_callback(pin, object, callback)
set_posedge_callback(pin, object, callback, tag)
set_negedge_callback(pin, object, callback);
set_negedge_callback(pin, object, callback, tag);
```

where:

`pin`	Specifies the pin of type `sc_in<T>`. For `set_posedge_callback` and `set_negedge_callback`, the pin has to be of type `sc_in<bool>`.
`object`	Is a pointer to the class containing the callback method.

`callback`	Is a pointer to a member function of the object class. It must have one of the following signatures:
	`void changeCallback()`
	`void changeCallback(int tag)`

`tag`	Is an user-provided integer that is passed to the callback.

Note

- The `SCML2_CALLBACK` macro can be used as a convenience macro for registering a member function as a callback.
- The Pin Callback functions can only be registered **before** end of elaboration, so for example in the module's constructor, but **not** in its `end_of_elaboration()` method.

4.2 Commands

This section describes:

- scml_command_processor
- scml_loader

4.2.1 scml_command_processor

`scml_command_processor` is an SCML object that provides the link between an interactive debugger and the simulation. It allows the debugger to execute commands in the simulation, for example, to switch the operating mode of a component or to generate data analysis about the execution and state of the model. The command processor operates local to a specific component in a model and can manage multiple commands each with their own set of arguments.

4.3 'Parameters

This section describes:

- scml_property
- scml_property_registry
- scml_property_server_if
- scml_simple_property_server

4.3.1 scml_property

`scml_property` is an SCML object used for model configuration, it can hold a value of type `int`, `bool`, `double` or `string`. The value of a `scml_property` is loaded during elaboration and is intended to provide the link between platform authoring tools and the simulation. It provides an easy to use and very flexible configuration mechanism for models. Typically, configuration parameters are stored in an VP Config which is generated by the authoring tool (Platform Creator) and is loaded by VP Explorer at the start of the simulation. Without recompiling or editing the model, it is then possible to change the configuration of the simulation simply by editing or modifying the XML file.

The following `scml_property` classes are available:

```
scml_property<int>
scml_property<unsigned int>
scml_property<double>
scml_property<bool>
```

```
scml_property<std::string>
scml_property<long long>
scml_property<unsigned long long>
```

The `scml_property` class is templated with the underlying value type. The following type definitions are available to support generic programming:

```
typedef T value_type;
typedef scml_property_base<value_type> this_type;
typedef this_type* this_pointer_type;
typedef this_type& this_reference_type;
```

The following constructors are available:

```
scml_property(const ::std::string& name);
scml_property(const ::std::string& name, T defaultValue);
```

where:

name	Specifies the name of the property. The name of the property is used together with the hierarchical SystemC name of the module to access the value of the property in the XML file.
defaultValue	Specifies the default value of the property. This default value is only used if the property is not found in the XML file.

The following assignment operators are available:

```
this_reference_type operator=(const scml_property<T>&);
this_reference_type operator=(value_type);
```

The following arithmetic assignment operators are available and behave as defined for the underlying value type:

```
this_reference_type operator += (value_type);
this_reference_type operator -= (value_type);
this_reference_type operator /= (value_type);
this_reference_type operator *= (value_type);
this_reference_type operator %= (value_type);
this_reference_type operator ^= (value_type);
this_reference_type operator &= (value_type);
this_reference_type operator |= (value_type);
this_reference_type operator <<= (value_type);
this_reference_type operator >>= (value_type);
```

A property object can be converted to the underlying value type:

```
operator T() const;
```

```
std::string getName() const;
```

Returns the name of the `scml_property`.

```
std::string getType() const;
```

Returns the type of the `scml_property`. The type can be one of the following strings: `int`, `unsigned int`, `bool`, `double`, `string`, `unsigned long long`, `long long`.

```
class mymodule : public sc_module
{
public:
  SC_HAS_PROCESS(mymodule);

  mymodule(sc_module_name name)
    : sc_module(name),
      intProp("intProp"),
      boolProp("boolProp"),
      doubleProp("doubleProp"),
      stringProp("stringProp")
  {
      SC_THREAD(my_thread);
  }

  scml_property<int> intProp;
  scml_property<bool> boolProp;
  scml_property<double> doubleProp;
  scml_property<string> stringProp;

  void my_thread () {
    cout << "mymodule: Int: " << intProp
    << " Bool: " << boolProp
         << " double: " << doubleProp
    << " and string: " << stringProp
    << endl;
  }
};
```

This module has a property of each of the possible types. Each of the properties is part of the initialization list of the constructor. The properties automatically get their value upon construction of the module. They can be used as their value types anywhere in the module.

4.3.2 'scml_property_registry

The scml_property classes can only be used inside SystemC modules. Using `scml_property` objects does not require any knowledge of the `scml_property_registry`. Objects of type `scml_property` get their values automatically.

Two mechanisms are available to load values in properties:

- An XML file exported by Platform Creator
- A custom property server

`scml_property_registry` offers an API to read the values of these parameters.

The following enumeration type is available:

```
enum PropertyType {
  GLOBAL,
  CONSTRUCTOR,
  MODULE,
  PORT,
  PROTOCOL
};
```

This enumeration type indicates the kind of parameter you want to access:

- `GLOBAL` is reserved for internal usage.
- `CONSTRUCTOR` indicates a constructor argument of a module.
- `MODULE` indicates a module parameter.
- `PORT` indicates a port parameter.
- `PROTOCOL` indicates a protocol parameter.

The `scml_property_registry` class is a singleton class. A reference to the instance of the class can be obtained by calling the static `inst()` function:

```
static scml_property_registry& inst();
```

The following functions are available for getting the values of a certain parameter:

```
int getIntProperty(PropertyType type, const std::string& scHierName,
                   const std::string& name);
bool getBoolProperty(PropertyType type, const std::string& scHierName,
                     const std::string& name);
std::string getStringProperty(PropertyType type,
                              const std::string& scHierName,
                              const std::string& name);
double getDoubleProperty(PropertyType type, const std::string& scHierName,
                         const std::string& name);
```

These functions all take the same parameters:

- `PropertyType` *`type`* specifies the type of property you want to access.
- `const std::string&` *`scHierName`* specifies the hierarchical SystemC name of the `sc_object` that contains the parameter. In case of a `PORT` or `PROTOCOL` property, this is the hierarchical SystemC name of the port. In case of a `MODULE` or `CONSTRUCTOR` parameter, this is the hierarchical name of the module.
- `const std::string&` *`name`* specifies the name of the property whose value you want to get.

```
bool setCustomPropertyServer(scml_property_server_if *);
```

Sets the custom property server.

Note This function must be called before any property that depends on it (to get its value) is constructed. For more information about the `scml_property_server_if` class, see ““scml_property_server_if” on page 108.

```
std::vector<std::string> getPropertyNames( const std::string& scHierName);
```

Queries the list of names of the available parameters.

```
scml_property_registry::PropertyType getPropertyType( const std::string& scHierName,
const std::string& propName);
```

Queries the type of property.

```
class myport : public sc_port<my_interface>
{
public:
  myport(const string& name)
    : sc_port<my_interface>(name.c_str())
  {
    // In a port, we can get our parameters by using the propertyAPI
    intParam = scml_property_registry::inst().getIntProperty
    (scml_property_registry::PROTOCOL, sc_object::name(), "intParam");
  }

  unsigned int intParam;
};
```

4.3.3 scml_property_server_if

This class defines the interface a property server should implement.

```
virtual long long getIntProperty(const std::string & name);
virtual unsigned long long getUIntProperty(const std::string & name);
virtual bool getBoolProperty(const std::string & name);
virtual std::string getStringProperty(const std::string & name);
virtual double getDoubleProperty(const std::string & name);
```

A property server should override these interface functions. It needs to provide the value of the property whose name is provided as the argument.

Default implementations are available. They return `0`, `false`, or the empty string depending on the type.

This example shows how to implement a custom property server, based on STL maps.

```
class exampleCustomPropertyServer : public scml_property_server_if {
public:
  exampleCustomPropertyServer() { this->load(); }
  virtual ~exampleCustomPropertyServer() {}

public:
  // scml_property_server_if
  virtual long long getIntProperty(const std::string & name);
  virtual unsigned long long getUIntProperty(const std::string & name);
  virtual bool getBoolProperty(const std::string & name);
  virtual std::string getStringProperty(const std::string & name);
  virtual double getDoubleProperty(const std::string & name);

private:
  // disable
  exampleCustomPropertyServer & operator= (const exampleCustomPropertyServer
                                                                          &);
  exampleCustomPropertyServer(const exampleCustomPropertyServer &);

private:
  void load();

private:
  // data members
  map<string, long long> mName2longLong;
  map<string, unsigned long long> mName2unsignedLongLong;
  map<string, bool> mName2bool;
  map<string, string> mName2string;
  map<string, double> mName2double;
};

void
exampleCustomPropertyServer::load()
{
  mName2string[ "HARDWARE.module1.myString" ] = "the string";
  mName2string[ "HARDWARE.module2.sub.myString" ] = "the string";

  mName2double[ "HARDWARE.module1.myDouble" ] = 1.234;
  mName2double[ "HARDWARE.module2.sub.myDouble" ] = 1.234;

  mName2bool[ "HARDWARE.module1.myBool" ] = true;
  mName2bool[ "HARDWARE.module2.sub.myBool" ] = true;

  mName2longLong[ "HARDWARE.module1.myInt" ] = 30;
  mName2longLong[ "HARDWARE.module2.sub.myInt" ] = -33;
}

long long
exampleCustomPropertyServer::getIntProperty(const std::string & name)
{
  const long long r = mName2longLong[ name];
  return r;
}

unsigned long long
exampleCustomPropertyServer::getUIntProperty(const std::string & name)
{
  const unsigned long long r = mName2unsignedLongLong[ name ];
  return r;
}

bool
exampleCustomPropertyServer::getBoolProperty(const std::string & name)
{
  const bool r = mName2bool[ name ];
  return r;
}

std::string
exampleCustomPropertyServer::getStringProperty(const std::string & name)
{
  const string r = mName2string[ name ];
  return r;
}
```

```
double
exampleCustomPropertyServer::getDoubleProperty(const std::string & name)
{
  const double r = mName2double[ name ];
  return r;
}
```

4.3.4 scml_simple_property_server

This class defines an example property server that implements the scml_property_server_if interface.

```
bool load(const std::string& fileName);
```

Loads the properties from the file. Returns `true` if the load succeeds, `false` if an error occurred.

```
virtual long long getIntProperty(const std::string& name);
virtual unsigned long long getUIntProperty(const std::string& name);
virtual bool getBoolProperty(const std::string& name);
virtual std::string getStringProperty(const std::string& name);
virtual double getDoubleProperty(const std::string& name);
```

Returns the value of the property. If the property is not found, a warning message is printed and a default value is returned.

The property files have the following syntax:

```
file ::= {line}*
line ::= typeLine | valueLine | commentLine
typeline ::= '[int]' | '[uint]' | '[bool]' | '[string]' | '[double]'
valueLine ::= name ':' value
commentLine ::= '#' string
```

where:

name	Specifies the hierarchical name of the property.
value	Specifies the value for the property.

Properties that appear before the first `typeLine` in the file are treated as `int` properties.

The following code shows an example of a property file.

```
intproperty: -1
[uint]
property1 : 0
property2 : 1234
[string]
property3 : This is a string property
```

4.4 Reporting

DEPRECATED: The `scml2_logging` stream output has been deprecated in Product Version M-2016.12 and redirected to the *FastTrack* infrastructure. However, you can temporarily enable the old behavior by using the environment variable `SNPS_VP_SLS_ENABLE_SCML2_STREAM_LOGGERS`.

This section describes:

- status
- stream
- severity

4.4.1 status

`scml2::status` is a simple object that maintains a string value and is used mostly to enable debugging and analysis features. The value of the object can be visualized in tracing views or be used for watchpoints and so on.

The `status` object is a very simple object that holds a status value in string format. It can be used as the base of other higher level modeling objects or it can be used to enable debugging and analysis for a module.

The include file of the status object is `scml2/status.h`.

The following sections describe:

- Constructors are available for the `status` object.
- Properties of the `status` object.

The following constructors are available for the `status` object.

`explicit status(const std::string& name)`

Creates a new status object with the specified name.

The following are the properties of the `status` object.

`std::string get_name() const`

Returns the name of the `status` object.

`void set_status(const std::string& status)`

Sets the new value of the `status` object.

`const std::string& get_status() const`

Returns the current value of the `status` object.

`const std::string& get_description() const`

Returns the description for the `status` object.

`void set_description(const std::string&)`

Sets the description for the `status` object.

4.4.2 stream

`scml2::stream` is a convenience object for logging. It allows a stream like formatting syntax to be used to log messages from models. The stream object will send the messages to backend logger objects for processing (for example, sending it to `std::cout`). The stream has a name and associated severity which allows to control the messages that are being logged, for example, to restrict them to a certain hierarchy in the design and or a severity level. There are predefined severity levels for error, warning, debug, note, and so on.

The include file for this object is `scml2/stream.h`.

The following sections describe:

- Constructor available for the `stream` object.
- Properties of the `stream` object.

The following constructor is available for the `stream` object.

`stream(const std::string& name, const severity& severity)`

Creates a new stream with the specified name and severity level. For information on severity, see "severity" on page 113.

`stream(const severity& severity)`

Creates a new stream with the specified severity level. For information on severity, see "severity" on page 113. The name of the stream will be the name of the current `sc_module`.

The following are the properties of the `stream` object.

`std::string get_name() const`

Returns the name of the `stream` object.

`const severity& get_severity() const`

Returns the `severity` object of the `stream` object.

`bool is_enabled() const`

Returns `true` if the `stream` object is enabled, or returns `false` otherwise. The `stream` object will be enabled in case at least one back-end object requests output from this stream.

All methods that are defined on `std::ostream` are also defined on the `scml2::stream` object. A `stream` object can be used as a replacement of an `std::ostream` object like `std::cerr` or `std::cout`. The `scml2::stream` object will send the output to the back-end logger objects only when the stream is flushed. This is done when `std::endl` or `std::flush` is written to the stream.

Note

`SCML2_LOG` can be used as a convenience macro while checking if a `stream` object is enabled.

For example:

```
SCML2_LOG(myStream) << "Debug output for myStream" << std::endl;
```

In case the stream is disabled, the macro will evaluate to one boolean check. There will be no performance impact caused by example, the operator `<<` or the implicit conversion operators in the debug output.

For performance reasons, the `SCML2_LOG` macro should always be used, or the `is_enabled` flag should be checked before sending output to the stream.

Similarly, the `SCML2_LOG_ASSERT` macro conditionally writes to a stream if its argument evaluates to `false`.

For example:

```
SCML2_LOG_ASSERT(value == 0x1234, mStream) << "Value can not be
0x1234" << std::endl;
```

4.4.3 severity

The `severity` object holds a severity name and value. Each `stream` object has an associated `severity` object. For information on the `stream` object, see "stream" on page 111.

Lower severity level values mean a higher severity.

The include file for this object is `scml2/severity.h`.

The following sections describe:

- Constructor is available for the `severity` object.
- Available properties of the `severity` object.
- Pre-defined severity levels.

The following constructor is available for the `severity` object.

```
severity(const std::string& name, unsigned int level)
```

Creates a new `severity` object with the specified name and severity level value.

The following are the available properties of the `severity` object.

```
const std::string& get_name() const
```

Returns the name of the `severity` object.

```
unsigned int get_level() const
```

Returns the severity level value of the `severity` object.

The following severity levels are pre-defined by the logging library:

- `internal_error (5)`
- `error (10)`
- `warning (100)`
- `note (1000)`
- `debug (10000)`

The `severity` object has the following static methods to create the predefined `severity` objects:

```
static severity internal_error()
static severity error()
static severity warning()
static severity note()
static severity debug()
```

4.5 FastTrack

FastTrack allows the modeler to trigger messages from within peripheral models. The following distinct severities are defined:

- `Error`
- `Warning`
- `Info`

Each severity has distinct categories. For a complete listing of the categories, see "FastTrack Categories" on page 116. '

The SCML2 modeling library provides macros to communicate a message to the FastTrack infrastructure, which will then present the information to the end user.

From the modeler perspective, the information that needs to be passed is limited to the severity, the category, and a user-defined message.

The information presented to the end user can assist both in flagging embedded software errors or modeling problems that are detected in peripheral models, and in debugging the problem that is identified.

The modeler is strongly encouraged to flag FastTrack errors or warnings from model code in those cases where the model would be able to detect fault conditions or potentially wrong programming actions.

Adding FastTrack logging messages to the models can aid both the modeler and the embedded software engineer in debugging embedded software or model problems when functional issues are identified during simulation.

Note Besides the messages that are explicitly inserted in model code by the modeler, SCML2 modeling objects will also trigger FastTrack messages when certain events occur (*implicit* messages). For a complete list of events that are logged by SCML2 modeling objects, see "Implicit FastTrack Messages" on page 120.

The subsequent section outlines the API available to the modeler. ''

This section describes:

- FastTrack API
- FastTrack Categories
- Implicit FastTrack Messages
- Suppressing FastTrack Messages

4.5.1 FastTrack API

The *FastTrack* feature is exposed to the modeler via the SCML2 convenience macros, which are defined in the header:

```
scml2/tagged_message_macros.h
```

Convenience macros for use in `sc_modules` (where the implicit 'this' pointer refers to an `sc_module`) are as follows:

- `SCML2_INFO` (`info` category)
- `SCML2_WARNING` (`warning` category)
- `SCML2_ERROR` (`error` category)
- `SCML2_MODEL_INTERNAL` (`model internal` category)

Convenience macros for use outside of `sc_modules` are as follows:

- `SCML2_INFO_TO` (module, `info` category)
- `SCML2_WARNING_TO` (module, `warning` category)
- `SCML2_ERROR_TO` (module, error category)
- `SCML2_MODEL_INTERNAL_TO` (`model internal` category)

where 'module' is a pointer to an `sc_module`. The FastTrack infrastructure will identify the message that is sent as originating from the given `sc_module`.

For a full description of `info`/`warning`/`error` categories, see "FastTrack Categories" on page 116.

The macros presented above can be treated as stream objects, making the addition of FastTrack messages to the existing or new models very easy and low effort.

For example:

```
SCML2_INFO(FUNCTIONAL_LOG)
<< "Initiating DMA transfer on channel " << mChannelID"
<< std::endl;
```

where the category `FUNCTIONAL_LOG` indicates to the end user that this message conveys information about the high-level behavior of the model.

An additional convenience macro `SCML2_ASSERT` will check a given condition, and in case the condition evaluates to `false`:

- A FastTrack error of category `FATAL_ERROR` will be raised.
- The simulation will be terminated.

This allows the modeler to check, report and exit on conditions which might lead to unstable or catastrophic behavior in the model.

For example,

```
SCML2_ASSERT(all_is_fine == true) << "Something very bad has happened" << std::endl;
```

Note

- The message content should be limited to a single line, and needs to be terminated with a `std::endl` character.
- Only one line should be sent to the `stream` object represented by the macro. For example, one should not write:

```
SCML2_INFO(FUNCTIONAL_LOG)
<< "Message 1" << std::endl
<< "Message 2" << std::endl;
```

- There is no need to include a timestamp or object name to the message, these will be added automatically.

4.5.2 FastTrack Categories

Based on the severity, the available FastTrack categories are `Error`, `Warning`, `Info`, `Model Internal`.

Each category has an `enum` value for use in the API while modeling, and a name which will be shown to the user when the message occurs.

The following categories are available for use with the `Error` severity:

Table 4-1 Error Categories

Category Enum	Category Name	Meaning
`GENERIC_ERROR`	Generic Error	Generic error raised by the model. Used as fall-back category for messages that are triggered by legacy SCML Stream or VRE logging infrastructure that is redirected to FastTrack.
`SCML_INVALID_API_USAGE`	SCML Invalid API Usage	Internal use by SCML2 modeling objects only. Points out wrong use of the SCML2 API.
`ACCESS_WRITE_RESERVED_VALUE`	Access Write Reserved Value	A register or bitfield is written with a reserved value.
`ACCESS_WRITE_UNDEFINED_VALUE`	Access Write Undefined Value	A register or bitfield is written with an undefined value.
`CONFIGURATION_ERROR`	Configuration Error	A parameter or combination of parameters on a peripheral model is set to an invalid value or combination of values.
`ACCESS_PERMISSION_CHECK_FAIL`	Access Permission Check Fail	A memory access to a register/bitfield or memory is not allowed. For example, due to insufficient privileges or due to the current configuration of the peripheral.
`ACCESS_WRITE_TO_READ_ONLY`	Access Write To Read Only	An attempt to write a read-only memory location is done.

Table 4-1 Error Categories

Category Enum	Category Name	Meaning
`ACCESS_READ_FROM_WRITE_ONLY`	Access Read From Write Only	An attempt to read from a write-only memory location is done.
`ACCESS_UNMAPPED_ADDRESS`	Access Unmapped Address	An attempt to access from an unallocated memory location is done.
`FUNCTIONAL_ERROR`	Functional Error	A peripheral model can use this to flag unexpected behavior in the model.
`FILE_NOT_FOUND`	File Not Found	A file is not found.
`FILE_FORMAT_ERROR`	File Format Error	A file has the wrong format or cannot be parsed.
`STATE_ILLEGAL_TRANSITION`	State Illegal Transition	A peripheral model can use this to flag an illegal state transition in a state machine.
`STATE_ILLEGAL`	State Illegal	A peripheral model can use this to flag reaching an illegal state in a state machine.
`BUFFER_UNDERFLOW`	Buffer Underflow	A peripheral model can use this to flag a buffer underflow, that is, an attempt to read from a buffer which no longer has sufficient active elements.
`BUFFER_OVERFLOW`	Buffer Overflow	A peripheral model can use this to flag a buffer overflow, that is, an attempt to write to a buffer that is already full.
`UNDEFINED_ERROR`	Undefined Error	An error which does not fit any of the other predefined categories.
`PROGRAMMING_ERROR`	Programming Error	A peripheral model can use this to flag a generic programming error, that is, it is being accessed by software in a way that violates the specification.
`FATAL_ERROR`	Fatal Error	Flags a fatal error condition, and will terminate the simulation in a clean way.

The following categories are available for use with the `Warning` severity:

Table 4-2 Warning Categories

Category Enum	Category Name	Meaning
`GENERIC_WARNING`	Generic Warning	General warning raised by the model. Used as fall-back category for messages that are triggered by legacy SCML Stream or VRE logging infrastructure that is redirected to FastTrack.
`SCML_IGNORED_CALL`	SCML Ignored Call	Internal use by SCML2 modeling objects only. Points out an API call without any further effect (for example, making the same call twice, or canceling an event which was not scheduled or already canceled).
`FEATURE_NOT_MODELED`	Feature Not Modeled	A peripheral model can use this to indicate that a given feature that is being exercised is not modeled and not planned to be modeled due to for example, the abstraction level.
`FEATURE_TBD`	Feature TBD	A peripheral model can use this to indicate that a given feature that is being exercised is not modeled yet.
`FILE_SW_IMAGE_OVERWRITE`	File SW Image Overwrite	During image loading, a previously written location gets overwritten.
`ACCESS_IGNORED`	Access Ignored	An access to a memory location is ignored or has no effect.
`ACCESS_INVALID_PERMISSION`	Access Invalid Permission	A warning raised when an access to a memory object is allowed, but is for example, ignored due to mismatching access permission settings.
`UNDEFINED_WARNING`	Undefined Warning	A warning which does not fit any of the other predefined categories.
`CONFIGURATION_WARNING`	Configuration Warning	Used to flag setting of parameters in a way that is not advised.
`PROGRAMMING_WARNING`	Programming Warning	A peripheral model can use this to flag a generic programming warning, that is, it is being accessed by software in a way that does not meet recommendations outlined in the specification.

The following categories are available for use with the `Info` severity:

Table 4-3 Info Categories

Category Enum	Category Name	Meaning
`GENERIC_INFO`	Generic Info	Generic information raised by the model. Used as fall-back category for messages that are triggered by legacy SCML Stream or VRE logging infrastructure that is redirected to FastTrack.
`FILE_OPEN`	File Open	Logs opening of a file.
`FILE_CLOSE`	File Close	Logs closing of a file.
`FUNCTIONAL_LOG`	Functional Log	Logs high-level functional behavior of a peripheral model, as an aid during debugging. The level should be such that it makes sense to a software developer.
`FUNCTIONAL_LOG_VERBOSE`	Functional Log Verbose	Logs detailed functional behavior of a peripheral model, as an aid during debugging. The level should be such that it makes sense to a software developer.
`FUNCTIONAL_LOG_INTERNAL`	Functional Log Internal	Logs internal behavior of a peripheral model, as an aid during (mainly) model debugging. This level of logging can expose implementation details of the peripheral model.
`STATE_TRANSITION`	State Transition	Logs a state transition of a state machine.
`CONFIGURATION_INFO`	Configuration Info	Logs parameter values/changes in configuration via for example, Tcl calls.
`SOFTWARE_LOG`	Software Log	Used for messages that are emitted directly by software running on a simulated processor, for example, using *SemiHosting*.

The following categories are available for use with the `Model Internal` severity.

Table 4-4 Model Internal Categories

Category Enum	Category Name	Meaning
`SCML_CALLBACK_ENTRY`	SCML Callback Entry	Internal use by SCML2 modeling objects only. Logs calling of a registered callback by an SCML modeling object.
`SCML_CALLBACK_EXIT`	SCML Callback Exit	Internal use by SCML2 modeling objects only. Logs returning from a registered callback by an SCML modeling object.
`LEGACY_LOGGING`	Legacy Logging	Logs emitted by legacy logging infrastructure like SCML stream.
`LEVEL0`	Internal Level 0	Lowest level log (least detailed) emitted by model code to aid model debugging.
`LEVEL1`	Internal Level 1	Emitted by model code to aid model debugging.
`LEVEL2`	Internal Level 2	Emitted by model code to aid model debugging.

Table 4-4 Model Internal Categories

Category Enum	Category Name	Meaning
`LEVEL3`	Internal Level 3	Emitted by model code to aid model debugging.
`LEVEL4`	Internal Level 4	Emitted by model code to aid model debugging.
`LEVEL5`	Internal Level 5	Emitted by model code to aid model debugging.
`LEVEL6`	Internal Level 6	Highest level (most detailed) log emitted by model code to aid model debugging.

4.5.3 Implicit FastTrack Messages

Implicit FastTrack messages are messages that are being emitted by SCML2 modeling objects. The subsequent sections outline which messages are being emitted, so the modeler is aware and does not add similar explicit messages in model code.

Note Messages that are emitted because of wrong API usage or configuration of SCML2 objects are not listed here.

The following table lists the implicit FastTrack `Error` messages.

Table 4-5 Implicit FastTrack Error Messages

Event	Category	Message
`Calling SCML2_ASSERT(true)`	`FATAL_ERROR`	*Message passed to* `SCML2_ASSERT`

The following table lists the implicit FastTrack `Warning` messages.

Table 4-6 Implicit FastTrack Warning Messages

Event	Category	Message
`memory_disallow_access_callback with disallow`	`ACCESS_INVALID_PERMISSION`	*[*`read`*/*`write`*] access denied at address [*`address`*] on [*`object`*]*
`memory_disallow_access_callback ignored access`	`ACCESS_IGNORED`	*[*`read`*/*`write`*] access ignored at address [*`address`*] on [*`object`*]*
`bitfield_disallow_read_access_callback ignored access`	`ACCESS_IGNORED`	*Read access ignored on [*`object`*]*
`bitfield_disallow_write_access_callback ignored access`	`ACCESS_IGNORED`	*Write access ignored on [*`object`*]*
`bitfield_disallow_read_access_callback with disallow`	`ACCESS_INVALID_PERMISSION`	*Read access denied on [*`object`*]*
`bitfield_disallow_write_access_callback with disallow`	`ACCESS_INVALID_PERMISSION`	*Write access denied on [*`object`*]*

Table 4-6 Implicit FastTrack Warning Messages

Event	Category	Message
`Enabling tracing on an scml_clock object`	`UNDEFINED_WARNING`	*Warning in* `scml_clock` *[*`object`*]: disabling optimizations, clock cannot be optimized when tracing*

The following table lists the implicit FastTrack `Model Internal` messages.

Table 4-7 Implicit FastTask Info Message

Event	Category	Message
Invoking a write callback on any `scml_memory` location or register	`SCML_CALLBACK_ENTRY`	*[*`callback function name`*] ([*`object name`*]): write to [*`address`*]: [*`data`*]*
Returning from a write callback on any `scml_memory` location or register	`SCML_CALLBACK_EXIT`	*[*`callback function name`*] ([*`object name`*])*
Invoking a read callback on any `scml_memory` location or register	`SCML_CALLBACK_ENTRY`	*[*`callback function name`*] ([*`object name`*])*
Returning from a read callback on any `scml_memory` location or register	`SCML_CALLBACK_EXIT`	*[*`callback function name`*] ([*`object name`*]) : read from [*`address`*]: [*`data`*]*
Invoking a write callback on any `scml_bitfield`	`SCML_CALLBACK_ENTRY`	*[*`callback function name`*] ([*`object name`*] : write [*`value`*] with mask [*`bitmask`*]*
Returning from a write callback on an `scml_bitfield`	`SCML_CALLBACK_EXIT`	*[*`callback function name`*] ([*`object name`*])*
Invoking a read callback on an `scml_bitfield`	`SCML_CALLBACK_ENTRY`	*[*`callback function name`*] ([*`object name`*])*
Returning from a read callback on an `scml_bitfield`	`SCML_CALLBACK_EXIT`	*[*`callback function name`*] ([*`object name`*] : read [*`value`*] with mask [*`bitmask`*]*

4.5.4 Suppressing FastTrack Messages

Suppressing a message means that it will not:

- be shown on standard output,
- be recorded to the analysis database, and
- trigger a `FastTrack` breakpoint.

To suppress specific FastTrack messages, place a comma-separated value (CSV) file called `suppressions.csv` in the simulation working directory.

The format of the CSV file is as follows:

- Each line represents one suppression rule.

- Each line consists out of the following comma-separated values, which reflect the values of attributes of the message that is to be suppressed. For a description of the meaning of the attributes, see the beginning of "FastTrack" on page 113. The attributes should be listed in the order as outlined in the following table:

Table 4-8 Sequential Order of the Attributes

Attribute	Optional (Y/N)	Regular Expression Support	Notes
Simulation Time	Y	N	unit is *ps* (pico seconds). This can be any one of the following: • a specific timestamp. • a single time range, two numerical values separated with a `-` character.
Severity	N	N	This can be any one of the following: • `error` • `warning` • `info` • `model internal`
Instance Name	Y	Y	
Category	Y	N	The possible values are contained in tables `5.2/5.3/5.4`. Do not use the enumerated values for the categories, but the category names as they appear in the *Details* view of the FastTrack event trace in Virtualizer Studio / VP Explorer.
Message	Y	Y	
Core Name	Y	Y	
Program Counter	Y	N	This can be any one of the following: • a specific program counter value. • a single program counter range, two numerical values separated with a `-` character.
Software Function	Y	Y	
Software Context	Y	Y	
Software File Line Info	Y	Y	

Note

- Fields that are left empty are treated as wildcards, meaning that the value of this specific field is not considered when determining when a message should be suppressed or not.
- Fields that support regular expressions will only be treated as such if prefixed with the string `regex:`.
- If a range is specified for a numerical field.
  - The lower value should be specified first.
  - The upper value is inclusive.

The following lines extracted from a CSV file illustrates the above:

- To suppress all messages of severity *error*, emitted by `HARDWARE.DISPLAY` if the program counter of the core that did the access that triggered the message is between `0x210` and `0x214` (inclusive):

  ```
  ,error,HARDWARE.DISPLAY,,,,0x210-0x214,,,
  ```

- To suppress all messages of severity *info*, emitted by `HARDWARE.DISPLAY`, if the message contains the string write:

  ```
  ,info,HARDWARE.DISPLAY,,regex:.*write.*,,,,,
  ```

- To suppress all messages of severity *info*, emitted by any module between `0` and `999999 ps`:

  ```
  0-999999,info,regex:.*,,,,,,,
  ```

- To suppress all messages of severity *info*, category `State Transition`, emitted by any module:

  ```
  ,info,regex:.*,State Transition,,,,,,
  ```

Chapter 5
Modeling Guidelines

This chapter describes:

- Requirements for a Virtual Prototype Model
- Virtual Prototype Model Content
- The SCML Modeling Guidelines for LT
- Synchronization and Modeling for Speed

5.1 Requirements for a Virtual Prototype Model

Virtual Prototype models are developed in order to provide software developers and integration engineers with an abstract model of the system. The goal is to enable them to create application, middleware, and/or driver software, to optimize software performance and to validate and optimized system and software architectures. The key requirements for a virtual prototype model to enable this use case are (taken from the TLM2.0 requirements specification):

- Running real unmodified software

  It is important that the object code as it will be compiled for the final system can be executed on the virtual prototype. This implies the use of Instruction-Set Simulators (ISSes) for the processors for which software will be developed.

- Simulation speed

  A virtual prototype is a model of a design that will be executed on a host machine. It is important that the virtual prototype can execute software at a speed that is as close as possible to real time. This for example means that it should be possible to boot an OS in a few seconds in order to support driver software development. At the same time, there is a trade-off between simulation speed and temporal accuracy, which implies that for use cases that require a higher level of timing accuracy there will be a speed penalty, although also here the goal should be to achieve the highest possible simulation speed.

- Register accurate

  In order to run embedded software correctly, the memory and memory-mapped register layout and content should be modeled.

- Functionally complete

  All consequences of the software interaction with the rest of the system should be modeled. This implies how software interfacing with memory-mapped registers influences their content or the content of other memory-mapped registers. It also implies to model the influence of interrupt signals and other sideband signals that have an effect on the execution of software.

- Loosely Timed (LT)

  Timing in a virtual prototype is intended to simplify the synchronization of hardware components with software. Timing information is not an indication of timing accuracy for the overall operation of the system. In an LT model, timer interrupts fire roughly at the expected time to successfully boot OS. In general it is important to have an indication of the speed of the hardware interactions with software (through interrupts, timers, and so on) and of how fast register content is updated. However, it is not required to have the exact timing for each and every event in the system in order to enable software development.

- Approximately Timed (AT):

  For the software optimization and architecture analysis use cases, the Loosely timed abstraction does not provide with sufficient temporal accuracy. In this case, it is required to add more timing detail to the models. This implies that timing details of the processor need to be modeled in the instruction set simulator, including its memory subsystem (caches, pre-fetch operations, and so on). The goal is that the resulting system provides with enough detail to derive reasonably accurate performance data to decide on optimization strategies, resource mapping and memory architectures.

- Debugging and analysis

  The virtual prototype should provide hooks to attach embedded software debuggers, and tools to perform software analysis for the design.

- Performance information

  It should be possible to derive reasonably accurate performance data from simulation to enable software performance profiling and optimization. For this, timing annotation information may need to be improved, which may lead to additional functionality of the system that needs to be modeled. For example, the caches and memory controllers of the system should now be modeled more accurately.

- Configuration

  Due to the possible speed difference when enabling performance information gathering, a runtime switch is required that enables these additional features.

5.2 Virtual Prototype Model Content

When assembling a virtual prototype, it is important to meet the requirements listed in "Requirements for a Virtual Prototype Model" on page 125. An actual embedded system usually consists of many components, not all of which need to be modeled in a virtual prototype. Some components have a direct relation to the execution of software, whereas others may have no relation to software. Only the components that are important to execute software correctly need to be modeled. Components like built-in self test, analog-digital converters, voltage regulators, on-chip debug interfaces, protocol converters, arbitration units, clock control, and any other block should not be modeled if they do not impact the functionally correct execution of software. A virtual prototype typically contains components of the following types:

- Processor cores

  To run the object code of the actual software, ISSes are used, wrapped into a module with sockets for the data communication, interrupt signals, and integrated into the multitasking kernel of SystemC. Processors that run software which is not part of the current development can be replaced with an abstract functional model of the software. Such a model contains the algorithm with timing annotation and explicit socket accesses to model the data communication with the rest of the system. When creating a virtual prototype, there is no need to model processors. Processor models are typically made available by the processor vendors.

- Memory and Interconnect Hierarchy IP

  Communication between different components in a system can be very complex, but depending on the use case it may not be necessary to model all the details of the system interconnect. Instead communication could be limited to reflect the memory-map decoding of the actual system. The data transactions in the model should include all information necessary for correct software execution. For example, information about secure accesses, protection, exclusive access should be modeled. TLM2.0 has provided generic interconnect models and has set the basis for other organizations to develop standard interfaces for industrial interconnect components. For software optimization and exploration use cases, the interconnect model should have sufficient timing detail to allow modeling of the protocol specific timing implications of data and instruction exchanges. Even more accurate models of the interconnect are needed when the focus of the architecture exploration focuses on the interconnect itself, these models should be reusable for the software centric use cases, but are generally replaced by their more abstract versions.

- Memory IP

  Obviously, the key component for software execution is to have a memory model. The memory subsystem does not have to include the full behavior of caches and memory controllers. It is possible to limit their function to have the control registers for these components modeled and leave the behavior out. However, if performance analysis of the software is required, then the functional behavior of caches and the memory subsystem should be present to be able to see the timing impact of the different data accesses by software. With SCML, generic memory models are trivial; so no special effort is required for these.

- Memory-mapped components

  For virtual prototypes, these components are very important: They allow the functional verification of the embedded software. Internal memory-mapped registers and the behavioral consequences of a register access should be modeled. Model creation for memory-mapped components is one of the key topics of this modeling guidelines manual.

- External interfaces

  USB, serial ports, Ethernet, audio and video IO, Camera, Firewire, SIM card and so on are key elements of many system on chip designs. For the virtual prototype development, they are partly memory mapped components, that is, the register interface of these components is important to enable embedded software development. However, in order to test the functionality of these components, the virtual prototype needs to be extended with a model of the 'external world' as seen by the system it is modeling. This external world can be modeled as real world IO by using similar capabilities in the host (for example, USB) allowing the platform to forward the communication to the host, alternatively virtual IO can be used where for example, the file system on the host is used to mimic things like `MMC`/`SD`/`SDIO`, `HDMI`, `SATA`.

- Communication components

  This refers to DMA, communication bridges between subsystems, shared memory components, accelerators, and so on. These components are important for the part of their behavior that changes the content of memories and memory-mapped registers or how these components can be accessed. As they have an impact on the execution of software, these components need to be part of virtual prototype models.

- Platform synchronization

  These components are required for scheduling and real-time software functionality and deal with synchronization in the system. Usually, these components are software configurable; for this part of their behavior they are not different from any other memory-mapped components. The difference is in the behavior of these components, where it is important to model the correct timing and synchronization of the system. These components are also addressed in this manual.

- Data processing

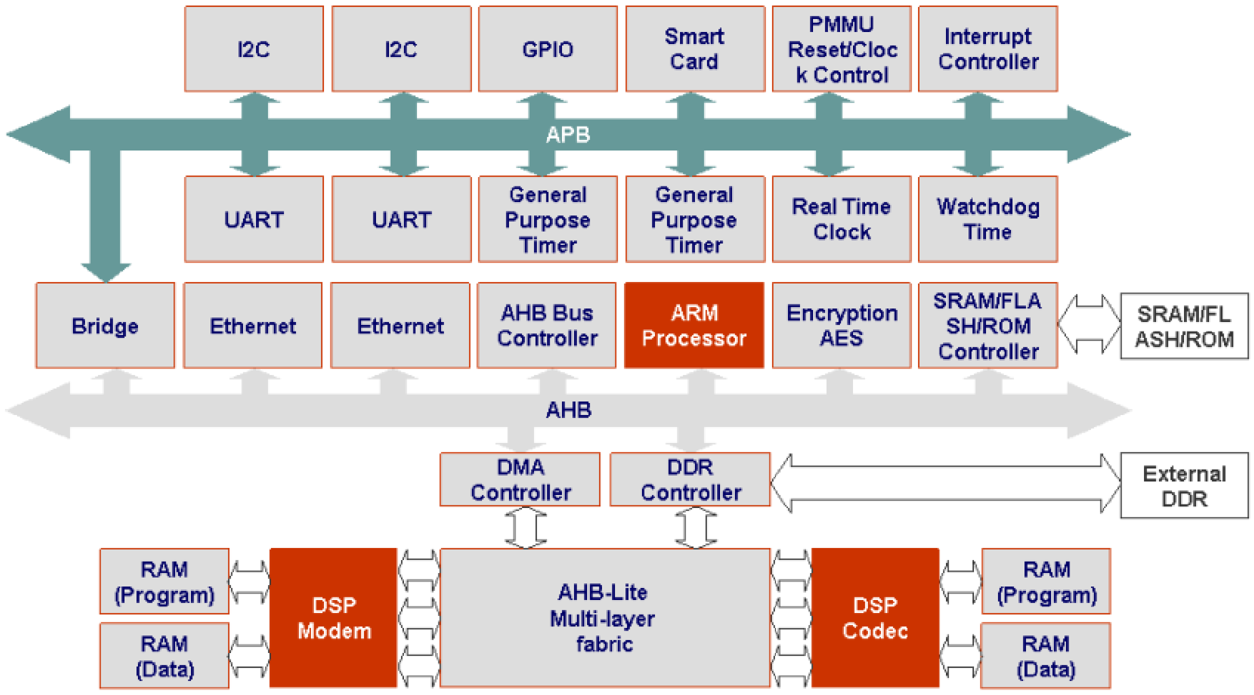
  These are all the blocks that can perform certain functionality related to the processing of data in parallel to the execution of software, but that are tightly controlled by it. For the virtual prototype use cases the exact implementation of the processing algorithm may be of lesser importance, the focus of the models for these components should be on the functionality they provide, their register interface and the mechanisms they use to exchange data with the embedded software. Hence, it is possible to develop these models reusing generic data processing libraries as they exist for the host and wrap them with register and data exchange interface models to include them in the virtual prototype.

- Subsystems

  As systems get more and more complex a key question when creating a virtual prototype is whether all components and subsystems need to be modeled for a certain use case or not. In general common sense should drive the decision when to stop. If the goal of the virtual prototype is to enable software development on the main application processor of the system, then it may not be necessary to model the wireless modem subsystem, or to have that limited to its data interface.

The following figure shows an example of a virtual prototype.

Figure 5-1 A Virtual Prototype



The purpose of SCML is to enable efficient modeling of virtual prototype components. This is achieved by encapsulating certain aspects of the TLM2.0 modeling standard with a generic implementation covering the most common uses of TLM2.0. SCML also provides a number of additional modeling objects that provide model-to-tool interactivity and a set of reusable timing objects.

5.3 The SCML Modeling Guidelines for LT

This section describes the rules to create component models for virtual prototypes using SCML. The modeling guidelines and coding style discussed here are focused on the loosely timed coding style. These rules are demonstrated through examples in the following sub-sections. All rules that are demonstrated in the examples are listed here. They are split into two lists:

- Modeling Methodology Guidelines
- Coding Style Guidelines

5.3.1 Modeling Methodology Guidelines

The generic SCML modeling guidelines for virtual prototype components are derived from the requirements to model virtual prototypes described in "Requirements for a Virtual Prototype Model" on page 125 and the coding style.

Specification documents for hardware components typically contain a lot of detail that is not important when creating a model that is going to be used in a virtual prototype.

The following table describes guidelines related to modeling only what you need.

Table 5-1 Rules for Modeling Only What You Need

Rule ID	Rule Description
1.1	**Only that part of the specification that is relevant to the execution of software should be modeled.** This is also called the *software observable state*. It is that part of the specification of the component that is accessible, visible from software, or that can have an impact on the execution flow of software. This means that the section describing the function of the component is important, as well as the part of the specification that describes the programming model with the detailed register layout.
1.2	**The detailed I/O interfaces should be abstracted into bus interfaces and modeled using TLM sockets.** virtual prototypes are mostly about memory-mapped communication between software and hardware. TLM2 sockets should be used to model these interfaces. Other interfaces should only be modeled when the activity on these interfaces influences the software execution. An example are interrupt interfaces.
1.3	**Where possible, timing information should be left out or abstracted to express the software execution rate.** There is no need to follow the detailed timing specification of the component. The most important reason to use timing information in a model is to model synchronization with the software and the rest of the system.
1.4	**Behavior must be implemented as a simple function call or possibly a state machine.**

The simulation performance of a virtual prototype model requires special attention. The goal is to get as close as possible to real-time execution for a model that will execute on a workstation. TLM2.0 defines the standard approach to create models for high-speed simulation. It is based on temporal decoupling and DMI (Direct Memory Interface). SCML enables these features in the SCML modeling objects so that these features are available by default. For more information, see "Synchronization and Modeling for Speed" on page 151.

The following table describes guidelines related to modeling for speed.

Table 5-2 Modeling for Speed Rules

Rule ID	Rule Description
2.1	**Temporal decoupling must be used for components that initiate transactions.** This typically relates to processor models. Temporal decoupling improves speed since it reduces the number of task switches between threads in a virtual prototype model. When all processor models would run in lock step, there would be a task switch every instruction or clock cycle; with temporal decoupling this will only happen every thousand or more instructions.
2.2	**Avoid the use of clock sensitive threads and methods.** This rule relates for example to timer components. When modeling a component that is only active irregularly or with large time intervals, it is better to use SCML clock objects or to use the SystemC event scheduler to schedule the exact time point where to execute the model behavior. Since task switches are expensive, it is important to avoid them; this also implies avoiding the use of threads and methods as much as possible.
2.3	**Minimize the number of transactions.** The number of transactions used to communicate a certain chunk of data determines the number of function calls used in the model (each transaction implies a blocking TLM interface). There is an overhead with each function call and payload that is created. By reducing the number of transactions, speed will be improved. For this reason, the TLM2.0 generic payload models bursts as a single transaction.
2.4	**Use DMI for all sockets that initiate transactions.** DMI is intended to reduce the impact of the infrastructure required to do communication between a processor and memory. By avoiding the assembly of a payload and the function call overhead to implement a data transaction, simulation speed is improved.
2.5	**Optimize for simulation speed when required.** Follow the strategy discussed in "Modeling Fast Target and Router Peripherals" on page 157. 1. Minimize synchronization by selecting the right synchronization level for memory-mapped register callbacks. 2. Optimize the most frequently accessed peripheral first. 3. Use read/write callbacks whenever it is possible to differentiate read behavior from write behavior. 4. Use dynamic registration and removal of callbacks for memory-mapped registers
2.6	**Apply C++ coding guidelines for speed.** All methods that improve the execution speed of software on a workstation apply here as well. 1. Reduce the number of data copies required to execute the behavior and communication of the model. 2. Avoid repeated external output (file access, terminal output). 3. Avoid repeated creation/destruction of complex data elements. 4. Avoid dynamic memory allocation

SCML provides memory modeling objects to model all memory-mapped storage and behavior. SystemC threads should be used for all independent and concurrent behavior in a system.

The following table describes guidelines related to modeling behavior.

Table 5-3 Modeling Behavior Rules

Rule ID	Rule Description
3.1	**Use the SCML memory modeling objects for all memory-mapped storage.** To separate the communication interface from the behavior, SCML provides memory, register, and bitfield objects. SCML memory modeling objects provide a default storage behavior for all accesses to the objects. These objects can interface with a socket of the module. So in a component model, each memory-mapped socket will have a memory object associated with it. The storage objects can be constructed in a detailed hierarchical model of the memory map of a component. The objects also implement the link to platform debug and analysis tools, and provide DMI access for initiators.
3.2	**Use callbacks on the SCML memory modeling objects for all memory-mapped behavior.** The default storage behavior of the SCML memory modeling objects can be overridden through callbacks. To model the impact of software accesses to the registers of a component, there are callback interfaces. Internal values of the component can be updated based on the information that comes with the register access. When a memory access results in communication to another component in the system, the behavior needs to synchronize the overall system state. For this purpose, callbacks can indicate their synchronization requirements when they are registered with the storage objects.
3.3	**Use SystemC threads for all behavior that is concurrent to the execution of software and that initiates memory-mapped communication.** In many cases, communication that is a consequence of a memory-mapped access can be modeled within the callback associated with the register access. In this case, all communication and behavior will be instantaneous for the software. Whenever the communication happens in the future or will be spread over multiple time points, a SystemC thread should be used. The thread will allow to model concurrent behavior to the execution of software.
3.4	**Use SystemC methods to model all behavior that is concurrent to the execution of software and that does not initiate memory-mapped communication.** SystemC methods are more efficient than SystemC threads for simulations speed. In a SystemC method it is not possible to use a quantum keeper or to initiate communication over a blocking interface like the TLM2.0 LT interface.
3.5	**Use a regular class method to model the state update for a component.** A typical behavior for a component is that its I/O and register values depend on the changes of both the I/O pins and the registers. This means that each time the value of one of the input pins or registers changes, all output should be recomputed. In such a case, it is best to create a recomputed method that is used both in the register callbacks as well as in the method sensitive to the input changes.
3.6	**Use submodules to model recurring behavior.** When a component is defined as a series of submodules with the same behavior, which together form a unique and well-defined component, it is best to create this as a single component for platform assembly. This implies that it is not necessary to use the usual SystemC or TLM communication mechanisms. Use `sc_module`s to define the submodules in order to ease debugging, so that naming of signals and internals get an additional SystemC hierarchy level. To communicate from the component level to the submodule, simple variable accesses and class method calls are sufficient. There is no need to introduce ports on the submodules or to use signals to communicate with the submodules.

Modules communicate using TLM interfaces. The goal of TLM modeling is to abstract the details of the pin interfaces between components in order to achieve higher simulation speed and make model creation easier. Virtual prototype modeling is based on the LT coding style of TLM2.0. Communication is done through a single function call from initiator to target, carrying the transaction payload and timing information.

The following table describes guidelines related to communication.

Table 5-4 Communication Rules

Rule ID	Rule Description
4.1	**Use the TLM2.0 generic transaction payload for all generic memory-mapped communication.** In SystemC TLM2.0, a generic payload is defined. It carries the typical information of a memory-mapped bus: address, data, data length, command (read/write), byte enables, response status. The SCML storage objects implement the behavior associated with this payload definition. The callbacks for memories can use the standard payload or also simplified information, in which case the memory objects will handle the semantics that are not forwarded to the callbacks. An adapter is provided that sits between the storage objects and the socket. This adapter implements some of the payload semantics that do not fit with the behavior of the storage objects (for example, separating burst accesses into individual accesses).
4.2	**Use a protocol-specific TLM2.0 payload when available.** When creating a model for a component that uses an interconnect interface for which a TLM2.0 LT payload or socket is defined, that specialized TLM2.0 payload has to be used. This requires that a dedicated protocol adaptor for the protocol is available in order to connect SCML memory objects to the specialized socket. An interconnect model using the specialized payload must be used to connect the components of the virtual prototype together.
4.3	**Use SystemC signals for all sideband communication between components.** For all infrequent communication between blocks that is not memory-mapped communication, SystemC signals can be used. It is important to make sure to synchronize before and after reading or writing to a signal from a memory-mapped behavior callback. This guarantees that the signal value that is used is set or seen at the right time in the system, so that the most up- to-date value is seen by all blocks involved.

The final foundation layer of the SCML modeling guidelines is timing. In a virtual prototype, timing information is used to model the synchronization rate of different components in a system. For details, see "Requirements for a Virtual Prototype Model" on page 125. Adding timing to a model can be done in various ways, but always implies to add more synchronization points into the model, which has an impact on simulation performance. For more information on synchronization, see "Synchronization and Modeling for Speed" on page 151.

The following table describes guidelines related to timing and synchronization.

Table 5-5 Timing and Synchronization Rules

Rule ID	Rule Description
5.1	**The timing parameter in the TLM2.0 LT interfaces must only be used for synchronization or to influence the software execution rate.** The TLM2.0 communication interfaces carry a timing parameter. This is used to annotate the timing of the data transfer. In a temporally decoupled system, it is the initiator that keeps track of its local time in relation to the quantum. So it will pass its current local time along with the transaction for the target to annotate the delay of the transfer. Whenever a target explicitly synchronizes with SystemC, it can use the value of the timing parameter to advance the global simulation time and set the timing annotation to `0` so that the initiator is aware of the synchronization that happened. A target can also use the value of the timing parameter to implement the delay of a software access to this memory-mapped location. This is not a measure for timing accuracy, but will influence the software execution rate of the software accessing the target versus other software that is not accessing this target.
5.2	**Increment the local time of an initiator with the execution time of the initiator on each transaction.** The initiators referred to in this rule are SystemC threads that initiate transactions independently of the software execution. As it is the initiator which the controls time in a virtual prototype, the local time parameter of the quantum should be incremented on every access. Before the TLM transaction is started, the local time should be updated to the point before the transaction. After the transaction, the time should be further updated with the timing for the current operation and next the quantum should be checked for synchronization (see 5.3). For example, an ISS would make sure local time is updated to the point the current instruction is started, do the TLM transaction next, and after the transaction increment the local time with the delay for the current instruction and then check for synchronization.
5.3	**There should be a check whether the local time is exceeding the quantum synchronization period after every transaction.** With every transaction, local time of an initiator can be further incremented by the target components. This will possibly cause an overrun on the quantum value; this should force synchronization with the rest of the system. This is done through the `need_sync()` and `sync()` API calls of the TLM2.0 quantum keeper. As explained in 5.2, the initiator also should take into account the delay for the current operation. This mechanism will ensure that any signal that is written can propagate to its final destination. **NOTE:** Whenever a signal is written, there may be multiple event-sensitive methods between the original source of the change and its destination. The delay after the transaction makes sure all these event-sensitive methods get triggered before the initiator starts another quantum.
5.4	**Use the SCML clock objects to schedule regular events.** The SCML clock objects are a more efficient implementation of clocks than the regular clocks found in the SystemC kernel. Avoid the use of SystemC clocks.

Table 5-5 Timing and Synchronization Rules

Rule ID	Rule Description
5.5	**Callback behavior on SCML memory objects must be set to `AUTO_SYNCING` whenever the correct execution of the behavior requires a synchronized system.** It is best to use `AUTO_SYNCING` in all cases and only change when optimizing for speed. Cases when `AUTO_SYNCING` is required include, but are not limited to: 1. The behavior queries the global SystemC time using `sc_time_stamp()` or `sc_simulation_time()`. 2. The behavior reads from an `scml_counter` object using `get_count()`, `read()`, or the `()` operator. 3. The behavior reads from or writes to an `sc_signal` or any other primitive SystemC channel (either internal to the component or when accessing a pin port). 4. The behavior notifies an event. 5. The behavior calls `wait()`. 6. The behavior accesses a port or socket that does not support temporal decoupling. This includes using the `post()`, `transport()`, `post_read()`, or `post_write()` methods of the `scml_post_port`. 7. The behavior models a software synchronization primitive, for example, when the storage is used for software semaphores.
5.6	**Use `SELF_SYNCING` behavior whenever the behavior complies with the following rules:** 1. The behavior is not `AUTO_SYNCING`. This means the behavior requires synchronization, maybe not with each invocation but at least in certain cases. 2. The behavior accesses ports or sockets that support temporal decoupling. This means they have a timing parameter according to the definition for TLM2.0 `b_transport`.
5.7	**Use `NEVER_SYNCING` behavior whenever the behavior will never require synchronization.** This requires that the behavior complies to both of the following rules: 1. The behavior is not `AUTO_SYNCING`. 2. The behavior is not `SELF_SYNCING`.

5.3.2 Coding Style Guidelines

This section describes guidelines related to the coding style.

The following table describes SCML-related guidelines.

Table 5-6 SCML-Related Rules

Rule ID	Rule Description
6.1	**A component must include the `<scml2.h>`, `<tlm.h>`, and `<systemc>` headers.**
6.2	**The template parameter of a `tlm_target_socket` and a `tlm2_gp_target_adapter` must be the same.** This means that both should have the same bit width defined.
6.3	**The size of the C++ data type specified as template parameter with an `scml::memory` object must be the same as the bit width of the adapter and socket it is connected to.**

Table 5-6 SCML-Related Rules

Rule ID	Rule Description
6.4	**For readability, it is advised to name the top-level SCML memory object `MemoryMap` and to put all internal registers of the module together in a `InternalRegisters` memory object.** This will improve the readability of the code, but also ensure that the representation of the model in debugging tools like VP Explorer is easy to use. All internal registers will be grouped next to the memory-mapped registers.
6.5	**Always make sure there is a visual link between the address values used by software and the index representation of `scml::memory`.** Addresses are specified as byte addresses in TLM2.0, while indices in an SCML memory are based on word size (for example, 32 bits, or 4 bytes). To avoid confusion over index versus address values, it is best to put the conversion calculation in the code. This can for example be done by specifying all index values as `address >> 2`.
6.6	**Use `scml2::initiator_socket` to model a socket through which software transactions or transactions concurrent to software execution will be issued according to the TLM2.0 generic protocol.** In case the transactions are not from a processor executing software, or are not concurrent to software execution, a regular TLM2.0 initiator socket can be used. This avoids an independent quantum for transactions going through this socket.
6.7	**Use an array of pointers to registers or memories when defining a range of registers or memories.** To make sure each register can be given a name and to make sure it is possible to easily iterate over a range of registers, you are advised to use a pointer to an array of registers or memories so that each individual register can be constructed and properly named.
6.8	**Whenever possible, use the same name for pins and sockets in the model as in the specification of the component.** This will improve readability and testing for the component model. This name should be used for the C++ object as well as the SystemC name passed in the initialization. Exception to this rule is when multiple objects would end up with the same name.
6.9	**Whenever possible, use the same names for memory-mapped registers and bitfields as specified in the specification of the component.** This will improve readability and testing for the component model. This name should be used for the C++ object as well as the SystemC name passed in the initialization. Exception to this rule is when multiple objects would end up with the same name. An example is bitfields that are specified with the same name as pins on a component.
6.10	**Use meaningful names for the callback methods.** When a method is only used as a callback, it should have a name that indicates when it will be called and for which register. In the examples of this modeling guidelines manual, methods are named according to the following naming template: `{on | post} {`*`registerName`*` | `*`bitfieldName`*`} {Read | Write | Transport}` For example, `onWdogValueRead` or `postWdogLoadWrite`.

Table 5-6 SCML-Related Rules

Rule ID	Rule Description
6.11	**Use meaningful names for the methods that will be made sensitive to events.** In the examples of this manual, methods are named according to the following naming template: `on`*`eventName`* or for signals: `on`*`signalName`*`Change` For example, `onTimeOutEvent` or `onIRQInchange`.
6.12	**Bit numbering in SCML registers.** When defining bitfields in a register, take into account that in SCML the Least-Significant Bit (LSB) is at position `0`. When your specification uses a bit numbering with the Most-Significant Bit (MSB) at location `0`, this should be taken into account.
6.13	**Reusing callbacks for multiple registers.** Use the tagged version of callback registration when the same behavior (or very similar behavior) is associated with a range of registers. This can be achieved by defining a memory alias for this range and register a tagged callback. The callback itself will have an additional parameter that indicates which specific register index triggered the callback. This allows to identify the register to be used in the behavior, or to differentiate parts of the behavior based on the register that was accessed.
6.14	**Do not duplicate state.** When registers, bitfields, or memory represent the state values of a component, it is not necessary to add separate variables in the model to represent the state. This leads to unnecessary copying and raises the risk of nonsynchronized values between the register interface and the internal variables.
6.15	**When the behavior of a register or bitfield access depends on the old and new value in the register or bitfield, use a regular read or write callback; do not use the `post_read` or `post_write` callbacks.** Since the `post_` callbacks store the new value before triggering the behavior, the old value is lost. Adding another variable to keep track of the old value is not necessary since the alternative of using a simple read or write callback avoids this. However, remember that a simple read callback does not store the value automatically, so this should now be part of the behavior.
6.16	**Use arrays or ranges to model a specification that defines `register_0...`*`n`*.** It is easier to work with `myRegister[0] ... myRegister[`*`n`*`]` than with `myRegister_0 ... myRegister_`*`n`*.

The following table describes SystemC-related guidelines.

Table 5-7 SystemC-Related Rules

Rule ID	Rule Description
7.1	**Always specify `SC_HAS_PROCESS(`*`className`*`)` for each component, either in the class definition or in the constructor.**

Table 5-7 SystemC-Related Rules

Rule ID	Rule Description
7.2	**Always make sure all SCML, TLM, or SystemC objects are given a name.** This name is preferably the same as the C++ object. This will improve readability and debugability of the code.
7.3	**When using `sc_core::sc_time` variables, it is best to construct these only once and reuse them in the component behavior.** Constructing `sc_time` objects is an expensive operation for speed. It is important to avoid constructing these objects each time you need them.
7.4	**To finish a simulation always use `sc_stop()`.** Never use `exit()`. The exception to this rule is the case that an error condition is triggered in the model that forces an immediate exit to avoid the model to core dump or to misbehave.
7.5	**Initialization of output ports should be done at end of elaboration.** Pin ports communicate through signals. These are separate objects that are instantiated separately. To be certain that the signal has been constructed, the initialization of output pins should happen at end of elaboration.
7.6	**Retrieve the clock period of clocks at or after the end of elaboration.** Clocks are connected through signals, similar to the input and output pins. This means the information about the actual SystemC clock or SCML clock that is connected to an input clock port is only known at the end of elaboration. It is best to store the clock period in an internal variable since the construction of `sc_time` objects is expensive; so this value can only be initialized at or after the end of elaboration.
7.7	**Writing to pins and signals from callbacks.** It is possible and advised to assign a value to an output pin or a signal from within a callback. There is no need to use a separate SystemC thread to do this. In RTL coding styles, signals and pins should be assigned from a single source to avoid multi-driver conflicts. The RTL simulator gives an error when multiple threads or methods assign the same signal. This ensures that in the actual hardware that is modeled, logic is present to decide what the final value for this signal should be. This rule does not apply in virtual prototype models and the simulator will not issue an error when this happens.

5.4 Synchronization and Modeling for Speed

The modeling style presented in this manual provides a standardized approach to create models for virtual prototype simulations. The intent of the guideline is to present an easy-to-use approach that delivers on the speed requirements for virtual prototype models, so that the developer can concentrate on making sure the model is functionally complete, has the right timing accuracy and is register accurate. However, it should be expected that simulation performance always needs to be a key consideration when creating models. In some cases however there may still be a need to trade-off performance for functionality. Besides that there is always a reason to look for further speed optimization. The goal of this section is to provide some background and guidance so that a virtual prototype model can be further optimized for speed.

- LT-Centric Simulation Techniques Overview
- Debugging Temporally Decoupled Systems
- Modeling Fast Target and Router Peripherals
- Optimizing Simulation Performance for FT Models

5.4.1 LT-Centric Simulation Techniques Overview

The first speed optimization available in TLM2.0 and the Synopsys SystemC simulation kernel is a Direct Memory Interface (DMI) infrastructure. This infrastructure basically provides the initiators with a direct *backdoor access* to all storage elements in the system (memories, registers). As explained in the *IEEE Std 1666* TLM-2.0 standard, an initiator can request to get immediate access to the storage within the target for simple storage accesses (read and write). The TLM2.0 infrastructure provides with means for the target to enable and disable this direct access. An interconnect component should forward the DMI requests.

In a virtual prototype model, it should be possible to configure the interconnect model to be as simple as an address decoder. Even in that case, the overhead of TLM function calls for communication is too big for high-speed simulations. So this is definitely an issue when using more refined bus models with more accuracy, for example, when adding more interconnect components to increase the accuracy of the functionally that is modeled or when adding more accurate timing information in the model.

So to have a configuration mode that takes full advantage of the speed optimization, the bypassed models (buses, transactors) should not consume any simulation time in idle mode. This can be achieved by an event-driven modeling style or by applying model gating, that is, disabling the clocks in idle mode, or as is done with the SCML2 clock objects: simply by de-registering the callback methods for the clock or by not triggering them.

At any time, it should be possible to switch between the speed-optimized simulation mode and the full simulation mode. The speed-optimized simulation mode uses the fast backdoor access to the target, while the full simulation mode simulates the complete detail of the bus transaction. It is only useful to switch from optimized mode to full simulation mode to run fast through some initialization sequence and then switch to a more accurate simulation mode. Switching from full mode to optimized mode is not useful since typically it is unsafe because of unfinished transactions stored in the buffers of buses and transactors.

The switch of the simulation mode can be initiated from the SystemC debugging tools VP Explorer and SystemC Shell:

- In VP Explorer

  Right-click on the system in the SystemC Design Browser (this is the root element for the design) and from the pop-up menu, select either *Switch to full simulation mode* or *Switch to speed-optimized simulation mode* (depending on the current simulation mode).

- In SystemC Shell:
  - To switch the abstraction level to full simulation mode, call:

    ```
    ::scsh::set_full_simulation_mode
    ```

  - To switch the abstraction level to speed-optimized simulation mode, call:

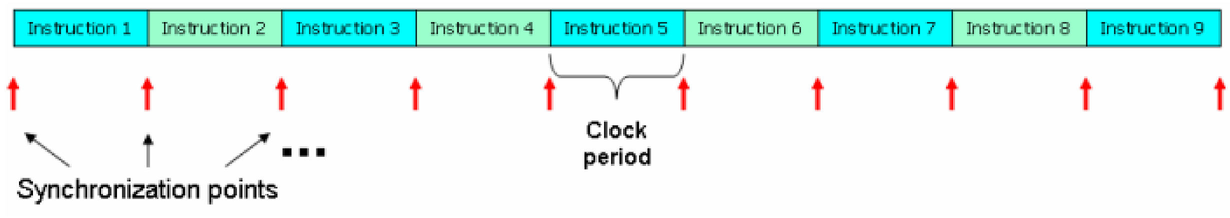
    ```
    ::scsh::set_speed_optimized_simulation_mode
    ```

On its own, the DMI would not have a big impact on the simulation speed. The full potential can only be realized in combination with temporal decoupling. Together, the two techniques reduce the interaction of the platform model with the SystemC kernel to the absolutely required minimum.

The DMI infrastructure described in "LT-Centric Simulation Techniques Overview" on page 151 provides a high-speed communication mechanism, which bypasses all SystemC interconnect models. This alone does not yield high simulation speed as long as the ISS is activated for every single instruction by the SystemC scheduler. In many cases it is not required for a platform simulation to have this rather accurate temporal granularity. A processor reads and writes to memory most of the time, without any impact on any other hardware block in the system, so there is no need to synchronize at every clock cycle, instruction, or transaction. The frequency with which software running on a processor synchronizes with the software on other processors or with the hardware in the system is much more infrequent than a clock cycle. To take advantage of this, *temporal decoupling* has been defined and standardized in TLM2.0.

A conventional way of building a SystemC simulation is to have a block synchronizing with the rest of the system at the granularity of a clock cycle or individual instructions. The benefit of this is that all internal state and I/O of all hardware blocks is synchronized at each clock cycle or each transaction. In the figure below this is shown by the red arrows.

Figure 5-2 Clock-Cycle-Based Synchronization as in Full Simulation Mode



A typical implementation of this is to have an `SC_METHOD` or `SC_THREAD` process, which is sensitive to an external clock signal. The process is activated by the SystemC scheduler on the raising edge of the clock signal, executes one control step of the ISS, and suspends. In the context of temporal decoupling, such an ISS is called fully synchronized with the SystemC scheduler, because the local time in the ISS is always identical with the global SystemC time. In general, a SystemC process synchronizes with the SystemC kernel by calling `wait()` in case of an `SC_THREAD` or `next_trigger()` in case of an `SC_METHOD`.

To improve the simulation speed, TLM2.0 standardizes temporal decoupling. This is implemented in the fast ISSes in the Synopsys Virtualizer. As shown in figure 5-7, the concept of a quantum is introduced, that is, the simulation kernel allows the ISS to run a certain amount of time without synchronizing with the SystemC scheduler. This means the SystemC thread containing the ISS executes so many instructions and only at the end of the quantum calls a SystemC wait. During the execution of such a quantum, the ISS basically runs ahead of the global SystemC time. The quantum in the example of figure 5-7 is assumed to be only four clock cycles, but the default duration in a real simulation is in fact 10,000,000 ns.

The *quantum* is a global value for the whole simulation and it is maintained by the quantum keeper. In many cases this could be a fixed value indicating the system-level synchronization for a certain platform and the software running on it. Therefore, it is possible to configure the value for the quantum for a certain simulation run. To improve on this concept, the Synopsys simulation engine introduces a quantum that is controlled by the simulation kernel. When starting a new quantum period, the kernel can make sure that the quantum will be smaller than or equal to the time to the next scheduled event in the kernel. This allows to automatically size the quantum with for example the period of a timer. On top of that, the quantum will terminate before the default duration is elapsed under the following conditions:

- The default duration of the quantum is expired. This is as in the default TLM2.0 definition.
- A SystemC event occurs.

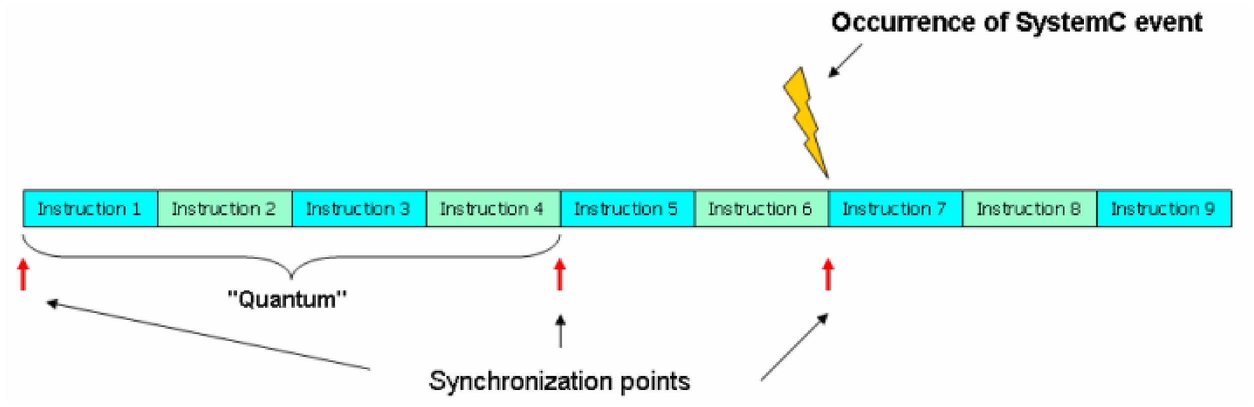
Note This refers to the point in time, where the SystemC kernel activates the process(es), which is (are) sensitive to the event. It does not refer to the notification of the event, that is, the point in time where a model calls the *notify()* method of the event (see B.70 and B.75 of [IEEE-1666]).

- A peripheral component, which requires synchronization, is accessed. This is as in the default TLM2.0 definition.

Whenever the quantum is "broken" by any of the above conditions, a new quantum will start with the default size or until the next known event. By the support to break a quantum whenever a SystemC event occurs, the Synopsys simulation engine removes the need to determine the size of the quantum before running the simulation. Typically, such a SystemC event is a timer interrupt being signaled or some other system-level event that otherwise should be calculated by the user and passed to the simulation.

The example shown in the following figure assumes that a SystemC event occurs after two cycles into the second quantum. This forces all ISSes in the platform to synchronize with the SystemC scheduler, because it may affect the software running on the ISSes. This may for example be the case if the event occurrence causes the firing of an interrupt. This way the quantum will dynamically resize to align with the system-level synchronization points.

Figure 5-3 Temporal Decoupling with an Occurrence of a SystemC Event After Instruction 6



The dynamic quantum is very important to enable a correct simulation. On the other hand, the breaking of the quantum by any event occurrence can have a serious impact on the simulation speed. One careless use of a clocked component forces the quantum's length to be one clock cycle, which completely ruins the simulation speed. You are strongly advised to use notifications of SystemC events with great care and avoid them whenever possible.

The synchronization property is specified with the callbacks of the registers. By default, a callback should be specified as `AUTO_SYNCING`. This means that before and after the callback the quantum will be broken, or in other words, the SystemC time and the local time will be synchronized (so local time is `0`; the initiator is no longer running ahead of the SystemC simulation). `AUTO_SYNCING` ensures that in a callback it is possible to:

- Query the SystemC time, for example by using calls to `sc_time_stamp()`
- Read and write SystemC signals
- Use the `notify()` methods of SystemC events

The synchronization ahead of the callback ensures that queries of the SystemC time return a correct value, as will a read from a signal give its most up to date value. By synchronizing after the callback, any writes to SystemC signals will be updated for all other initiators to see. The same is true for the event notification.

Alternatively, it is possible to set the synchronization of a callback to `SELF_SYNCING`. In this case, no automatic synchronization will be implemented. It is possible to implement the synchronization at any point in the callback by calling `wait()` with the local time as parameter. This enforces synchronization between SystemC time and local time; it is important to reset the local time parameter to `0` after such a call. The local time is the parameter that is passed as an argument to the transport and read/write callbacks.

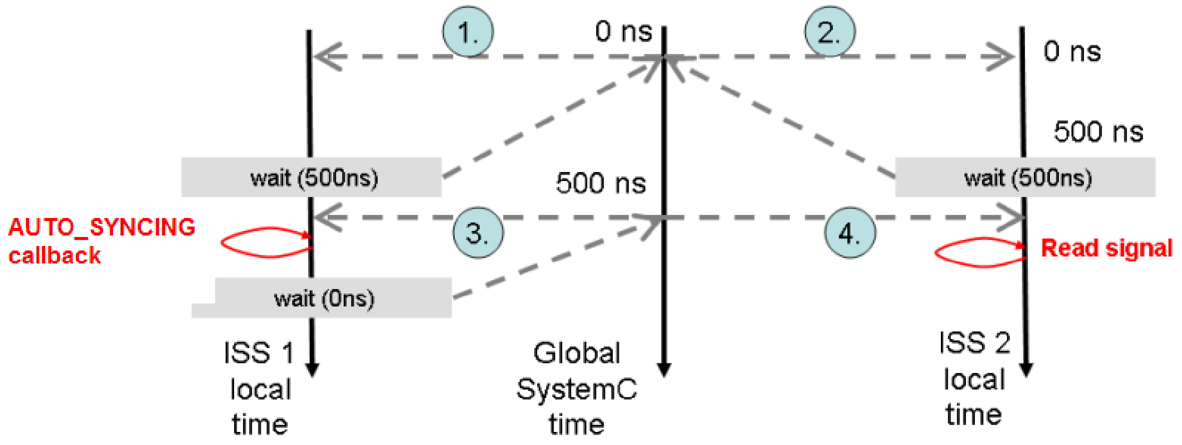
When no synchronization is needed at all, the synchronization option NEVER_SYNCING can be used to provide better readability of the code.

5.4.2 Debugging Temporally Decoupled Systems

This section discusses temporal decoupling by means of example scenarios.

Special care is required when using temporal decoupling in a multiprocessor system. This is illustrated in the example shown in the following figure, assuming a default quantum that is much larger than 500 ns.

Figure 5-4 Temporal Decoupling in a Multiprocessor System: Scenario 1



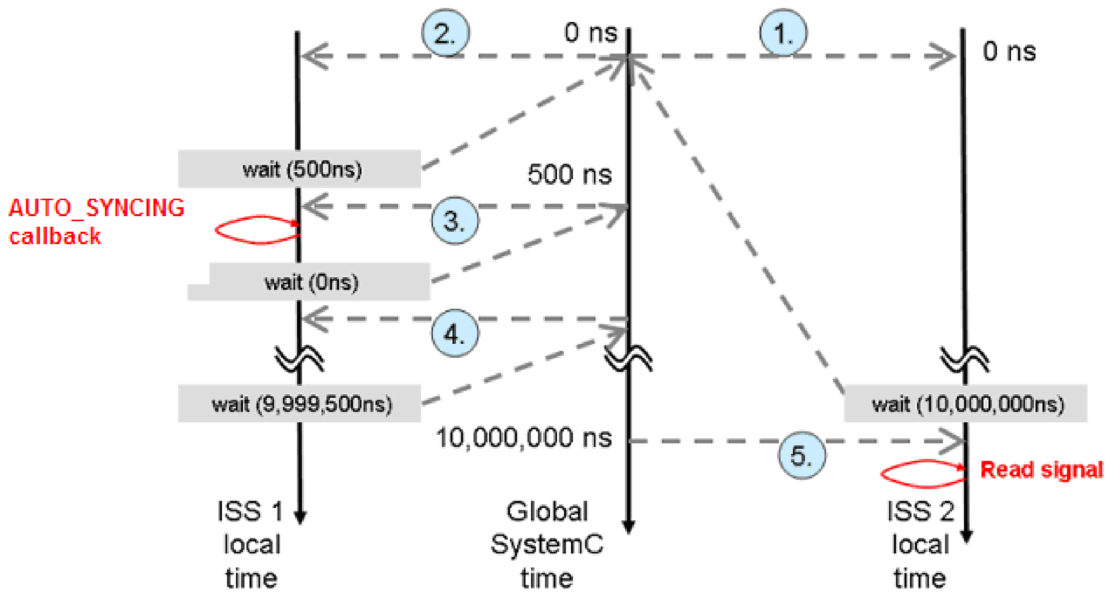
As shown in the above figure:

1. ISS1 executes for 500 ns in a temporally decoupled mode and then triggers a hardware block that wants to write an interrupt for ISS2 in an AUTO_SYNCING callback. As a result, the quantum for ISS1 terminates before the access to the hardware block happens and ISS1 synchronizes by calling `wait(500ns)`.
2. The SystemC kernel immediately activates ISS2 with a reduced quantum of 500 ns (there is an event scheduled at 500 ns; so the quantum size is limited to that point). The ISS runs in temporally decoupled mode and then synchronizes.
3. After 500 ns, the SystemC kernel reactivates ISS1, which performs the access to the peripheral in the AUTO_SYNCING callback. The behavior inside the callback writes the SystemC signal and after the callback, ISS1 again terminates its quantum by calling `inc()` and `sync()` (which amounts to a `wait(sc_zero_time)`).
4. The SystemC kernel activates ISS2, which handles the signal at the correct time and then executes for the duration of the next quantum.

In this case, everything works as expected. However, the sequence of activation between ISS1 and ISS2 is nondeterministic. Therefore, there is a 50% chance that the ISSes are activated in the opposite order.

The following figure illustrates the scenario where ISS2 gets activated first.

Figure 5-5 Temporal Decoupling in a Multiprocessor System: Scenario 2



As shown in the above figure:

1. `ISS2` executes for the duration of the default quantum and then synchronizes with SystemC time. The default quantum is `10,000,000ns`.
2. `ISS1` executes for 500 ns and then triggers the `AUTO_SYNCING` callback. Like in figure 5-8, `ISS1` synchronizes by calling `wait(500ns)`. However, in this case `ISS2` has already executed up to 10,000,000 ns, so its next activation will continue from that point forward.
3. After 500 ns, the SystemC kernel activates `ISS1`, which performs the access to the peripheral with the `AUTO_SYNCING` callback. The behavior of the callback writes to the SystemC signal and `ISS1` immediately yields to the SystemC kernel.
4. `ISS2` remains in waiting mode, so `ISS1` is reactivated and executes for the duration of the default quantum.
5. At 10,000,000 ns, `ISS2` is reactivated and only now it sees the interrupt signal.

In this second example, `ISS2` reacts too late to the interrupt signal from `ISS1`. As a result, the system may not behave as expected, although the platform model as well as the software running on the platform is correct. In this case, there are a couple of ways to address this issue:

- Reduce the default size of the quantum. This is the brute-force approach. The quantum can be reduced to `0` to make sure the two cores run in lock-step. At this point, the software in the example above would run correctly. Obviously, a trial-and-error approach to find a quantum value that delivers maximal speed can be used as well. Even in the case the quantum is set to zero, it is possible that software that runs correctly on a real system does not run on the virtual prototype. This could be due to the relative speed of software on different cores. On average, one core could be executing more instructions per cycle than another. This could be due to architectural differences between the two cores, but also due to differences in the memory subsystem for each core (caches, interconnect, memory controller).
- When a problem is identified to be caused by quantum effects due to scheduling order, it is possible to use the virtual prototype debugging tools to enforce synchronization only for the points where the interrupt will be set. In other words, breaking the quantum for those points in the simulation where this

matters. This can be done by setting a breakpoint on the simulation time where the interrupt will be set. In the Synopsys simulation environment, a breakpoint will cause the simulation to synchronize. This breaks the quantum.

If the exchange would not be through interrupts but in case `ISS2` reads a signal through some software access, then the peripheral access in `ISS2` is by default `AUTO_SYNCING`. This means that it will force the quantum to break at the point the software reads the signal. There would be a synchronization before the callback is called. This would allow `ISS1` to run forward to the same time point. `ISS1` will set the signal through another `AUTO_SYNCING` callback which will make sure the updated signal value is available before `ISS2` continued with its own callback.

If these accesses are not for signals but for variables in a memory it does not make sense to make all accesses to that memory `AUTO_SYNCING`. In that case it is better to reduce the size of the quantum. However, such a case points to an unsafe approach to synchronization between the two software applications on these processors. In real life it is very hard to predict that a variable access will happen exactly at the time required to ensure write-read order. Typically, such variables should be guarded by a mutex or semaphore, which in turn will typically rely on interrupt signals to ensure safe access. If that cannot be done it is better to try to make all accesses to the memory region that contains the semaphores `AUTO_SYNCING` accesses.

Unfortunately, some of the issues introduced by temporal decoupling are hard to pin down.

For example, a callback which is incorrectly registered as `NEVER_SYNCING` can be easily overlooked. In fact, the simulation may still behave correctly until a change in the platform or a change in the random order of initiator activations suddenly unearths the problem. In this situation, the problem is typically very hard to detect since the occurrence is completely unrelated to the actual reason. In the same way, the multiprocessor corner case described in “Debugging Temporally Decoupled Systems” on page 154 can be hard to make out.

Still, there are several strategies to detect simulation issues due to temporal decoupling:

- Check that all behavior registered through callbacks on memory-mapped devices is set to `AUTO_SYNCING`. Revert all suspicious `NEVER_SYNCING` or `SELF_SYNCING` accesses to `AUTO_SYNCING`. This will slow down the simulation performance of the platform, but may point out the synchronization problem that exists in the model.
- A simple approach that does not require to inspect every callback in the system is to simply run the simulation with quantum set to zero. If this works correctly, there is a good chance that there is some synchronization problem. The simulation performance impact of this approach is, however, huge.
- Each temporally decoupled core provides a method called `show_optimized_accesses()`. This gives you the complete list of memory regions, which are accessed through DMI.
- Running the simulation with quantum set to zero sometimes takes too long. In this case, the default quantum period can be gradually reduced. This can be done in VP Explorer using the `set_quantum` command. For detailed information, see "set_quantum" in the *VP Explorer Tcl Interface Reference Manual*.
- Another approach in case running the system with quantum set to zero takes too long is to selectively disable optimized accesses for a suspicious region. This can be done in VP Explorer. To do so, right-click on a memory in the Register view and then select *Backdoor Access* from the pop-up menu to disable it. For detailed information, see "Controlling the Simulation Mode and Backdoor Accesses" in the *VP Explorer User's Guide*.

5.4.3 Modeling Fast Target and Router Peripherals

This section discusses the detailed modeling guidelines for target and router peripherals.

As explained in “LT-Centric Simulation Techniques Overview” on page 151, the type of the callback on an SCML target object (`memory`, `reg`, `router`) determines the simulation speed overhead for accessing this

object. Depending on the frequency of accesses, a too conservative callback can seriously limit the overall simulation speed.
The following table associates rough speed level with each type of peripheral access. The three right-most columns give the functional and synchronization features of each type of access that determine the speed impact. As the speed impact increases, the speed-level numbering increases.
The "Synchronization over the Bus" column refers to the type of interconnect model used. By preference, virtual prototypes use an LT memory-mapped model to interconnect the peripherals to the initiator. An LT interconnect is a simple address decoder without any additional synchronization, whereas AT or cycle-accurate interconnect models infer some or many additional synchronization points in the simulation to model for example arbitration.

Table 5-8 SCML Access Levels

Speed Level	Type of Access	Synchronization over the Bus	Synchronization Required	Behavior Attached to the Memory
1	DMI pointer access	No	No	No
2	`NEVER_SYNCING` callback	No	No	Yes
3	`AUTO_SYNCING` callback	No	Yes	Yes
4	Bus Access over non-LT bus	Yes	Yes	Yes

- Speed level 1 is by far the fastest peripheral access. In this case, the initiator gets access to the internal storage of the peripheral through the DMI APIs and no user-defined behavior is invoked. This corresponds to a target object without callbacks. The speed is not affected by the number of interconnect nodes or the type of interconnect node; also the router objects do not have an impact on this. As long as the DMI request is enabled throughout the whole path from initiator to target object there will be a direct access to the storage of the target object.
- Speed level 2 is the case for a `NEVER_SYNCING` callback where no additional synchronization points are set inside the behavior. `NEVER_SYNCING` is in essence the same as `SELF_SYNCING` but through the name the user shows the intent that no `wait()` calls will be used in the implementation. In this case, where the interconnect model is still an LT address decoder, no synchronization is implemented for the peripheral access. Since behavior is attached to the target object, it is required to create an TLM2.0 payload and a set of TLM calls need to be made to get to the target peripheral. As a result, the access to a level-2 peripheral is up to 50 times slower than accessing a level-1 storage object.
- Speed level 3 refers to peripherals with a `AUTO_SYNCING` callback attached to the target objects. The access to a level-3 peripheral requires synchronization before and after the call is made. Each synchronization infers a context switch of all ISS initiators in the system, so the simulation speed is another factor 20 slower than at speed level 2.

Note This is the default and preferred setting for callbacks on target objects. Therefore, it is important to investigate whether some of these can be reverted to speed level-2 accesses.

- Speed level 4 refers to virtual prototype models where the interconnect model is not a simple LT address decoder but a more complex AT or cycle-accurate bus model. To increase the timing accuracy of these models, additional synchronization points are implemented inside the bus model so that an access from initiator to target will infer a multitude of synchronization points. These typically will have more speed impact than the additional synchronization of an `AUTO_SYNCING` callback. Such a model will be another factor 10-100 slower than the speed level-3 model. Therefore, it is very important for fast virtual prototypes to use an LT address decoder instead of an more accurate bus model.

Obviously, there are many peripherals in a complete virtual prototype model and the impact of an individual peripheral should be weighted with the frequency of accesses to that peripheral. Imagine the case of a platform model with an accurate bus that supports DMI and where only one out of one million accesses are done over the bus (so all other accesses are DMI pointer accesses). Then the speed impact of that single level-4 access will still bring down the total simulation speed significantly but when compared to a platform where all accesses are level-4 accesses it will seem infinitely faster.

As described in "LT-Centric Simulation Techniques Overview" on page 151, callbacks on target objects are by default `AUTO_SYNCING`. This modeling guidelines manual promotes to use this as a default since it is the safest approach to get the expected behavior. At the same time, "LT-Centric Simulation Techniques Overview" on page 151 shows that a `NEVER_SYNCING` callback has much better simulation performance. The main benefit of an `AUTO_SYNCING` callback is that it ensures that all initiators in the system have had the chance to catch up to the current local time in the running initiator, and that SystemC time is updated to the current initiator time (that is, local time is now `0`). There is still the chance that one or more initiators are further ahead in their local time.

5.4.3.1 AUTOSYNCING

In general, it required to stick to `AUTO_SYNCING` callbacks whenever the correct execution of the behavior requires a synchronized system. In the following cases, a `AUTO_SYNCING` callback should be used:

- The behavior queries the global SystemC time using `sc_time_stamp()` or `sc_simulation_time()`. In case `NEVER_SYNCING` would be used, these functions would return the SystemC time at the beginning of the current quantum (that is, they do not take local time into account).
- The behavior reads from an `scml_counter` object using `get_count()`, `read()`, or the `()` operator. Since the return value of these methods is computed based on the global SystemC time, the effect would be the same as in the previous point.
- The behavior reads from or writes to `sc_signal` or any other primitive SystemC channel. Writing to a primitive channel in a `NEVER_SYNCING` callback would only update the projected value. As a result, readers of the primitive channel would only see the new value after the `update()` function of the primitive channel has been called by the SystemC kernel, which is only after the quantum has expired.
- The behavior notifies an event. This operation updates the state of the event queue in the SystemC kernel and hence requires the current thread to yield (that is, call `wait()`). Only then will the SystemC kernel process the notified event correctly. Same as above, an event notification inside a `NEVER_SYNCING` behavior would only be taken into account after the current quantum expires.
- When the behavior has synchronizing semantics, an example is a semaphore register. At first glance, the default behavior of an `scm2::reg` perfectly models a register. However, the semaphore semantics of this register require that every access to this register happens at the correct SystemC time, so that other accesses to the semaphore are correctly handled.
- The behavior uses the `post()`, `post_read()`, or `post_write()` methods of the `scml_post_port`. This rule is a consequence of the rule on notifying an event since the implementation of these methods notifies the `end_event` of the `scml_transaction` object.

- The behavior calls the `transport()` method of `scml_post_port`. Unlike the `post_` methods listed above, the `transport()` method of `scml_post_port` does not notify any events and does not have any other side-effects. However, any component called through this `transport()` method could have synchronizing behavior and since the `transport()` call does not have a timing parameter, it is required to synchronize with the rest of the system before calling transport.

5.4.3.2 SELF_SYNCING

It is safe to use `SELF_SYNCING` callbacks in the following cases:

- The behavior calls `wait()`. By calling `wait()`, the behavior effectively has become a synchronizing behavior itself so there is no need to use `AUTO_SYNCING`. Whenever `wait()` is called, the local time should be passed as an argument to the `wait()` call to make sure local time and SystemC time are correctly aligned. The timing argument of the wait call may further be incremented with any additional delay that needs to be modeled. After the `wait()` call, the local time parameter should be reset to `0`.
- When the behavior calls the `b_transport()` call on an initiator socket. As the timing parameter will be forwarded, any behavior further in the call chain can correctly synchronize.
- With any APIs that take a timing parameter to support temporal decoupling. These could be available from convenience sockets that come with extended TLM2.0 protocol definitions.

As mentioned before, in case of doubt, a callback should be registered as `AUTO_SYNCING` to ensure the correct execution of the functionality.

5.4.3.3 NEVER_SYNCING

Use `NEVER_SYNCING` behavior whenever the behavior will never require synchronization. This requires that the behavior complies to both of the following rules:

- The behavior is not `AUTO_SYNCING`.
- The behavior is not `SELF_SYNCING`.

For example:

- The behavior does not use any SystemC constructs and does not have synchronizing semantics. This applies to all purely arithmetic blocks like hardware accelerators.
- The behavior only accesses `scml2::memory` or `scml2::reg` objects.

This section discusses a number of additional strategies to optimize for simulation speed.

- Optimize the most frequently accessed peripheral first.

  This means that the memory that is accessed most frequently should have the lowest speed level. The practical application of this simple rule requires some knowledge about the system and should be generalized to all accesses to a peripheral.

  A simple example illustrates this idea: Imagine a peripheral that has three registers: `A`, `B`, and `C`. The behavior of the peripheral is that register `C` always contains the sum of `A` and `B`. There are two options to model this:

  - This can be modeled by adding a `NEVER_SYNCING` write callback to registers `A` and `B` so that on every write to these registers the value of `C` is updated. Register `C` does not need a callback.
  - An alternative approach is to have a `NEVER_SYNCING` read callback on `C` where every time `C` is read, `A` and `B` are added and stored in `C`.

  Depending on the number of times `A`, `B`, and `C` are read or written, one alternative is better than the other. This decision may depend on the system context.

- Use read/write callbacks whenever it is possible to differentiate read behavior from write behavior.

  The SCML memory object offers the following kinds of callbacks:

  - The transport callback gives access to all attributes in the TLM2.0 generic payload.
  - The read and write callbacks are essentially a convenience shortcut which give access to a limited set of attributes.

  There is basically no difference in simulation speed if you use either a transport callback or both read and write callbacks of the same speed level. A speed advantage can be achieved if the speed level for read and write can be different. For example, if a component behaves as plain storage for read accesses but for write accesses there is some behavior implied then it is better to implement a write callback if possible. Using a transport callback to model such a component would be unnecessarily slow. Only having a write callback ensures that DMI is still enabled for read accesses.
  All default behaviors (as `write_only`, `ignore_access`, and so on) are also implemented using callbacks with `NEVER_SYNCING` synchronization.

- Use dynamic registration and removal of callbacks.

  One more strategy to optimize simulation speed is to adjust the speed level of the callback during simulation time. Callbacks can be registered to change from `NEVER_SYNCING` to `AUTO_SYNCING`, for example, when the register models a serial buffer that needs to synchronize with the rest of the system every 512 accesses. In this case, the callback will keep track of the number of accesses and switch itself to `AUTO_SYINCING` when the update will need to happen.

Note Registration and removal of callbacks are in itself expensive operations so a careful trade-off has to be made.

5.4.4 Optimizing Simulation Performance

This section provides an overview of the simulation infrastructure.

The SCML Modeling methodology is built on top of SystemC. SystemC is a C++ based modeling library with a cooperative event-driven process scheduler at its core. This means that at the core SystemC provides with deterministic mechanisms to model the details of the concurrent hardware. The drawback of these mechanisms is that they rely on a scheduler and process context switches. Hence, have a very negative impact on simulation performance. Typically, the simulation speed of a fast stand-alone instruction-accurate ISS drops by more than one order of magnitude when it is integrated into a SystemC platform.

Over time, several extensions have been added to SystemC, primarily to circumvent the process and event based modeling basics of SystemC. This is done to improve the simulation performance so that SystemC-based solutions can keep up with the ever increasing complexity of hardware systems. In the first step, SystemC has been extended with *Interface Method* calls for communication, which is better known and further standardized as Transaction Level Modeling (TLM). This replaces signal and event based communication between models with interface method calls. This increases the risk for non-determinism and the ugly debugging problems that come along with it, but this risk is low compared to the simulation speed benefit.

In the next step the synchronization requirements to model communication between components is further reduced. In the TLM2.0 standard, the concepts of temporal decoupling and direct memory interface have been introduced.

Using a TLM modeling style with function calls for communication with the platform context of the ISS is not sufficient. The overhead incurred by requiring TLM communication for every data exchange by an ISS limits the reachable simulation speed. To enable the use of SystemC-based platforms for software development use cases, TLM2.0 has standardized two major optimization techniques, which together enable simulation speed of 10-500 MIPS and beyond for an entire SOC platform. The two optimizations are:

- DMI infrastructure

  Direct Memory Interface (DMI) forwards memory requests directly from the initiator to the target. This technique - also known as *backdoor access* - bypasses the complete interconnect model, which typically comprises components like buses, bridges, and transactors.

- Temporal decoupling

  Temporal decoupling reduces the interaction between the platform simulation and the SystemC scheduler. This is achieved by drastically extending the synchronization interval between the model and the simulation kernel. A traditional model is activated by the SystemC kernel at the level of cycles or transactions. A temporally decoupled model synchronizes with the SystemC kernel at the level of *quantum*s, which can last as long as 100,000 cycles.

Taken together, the two techniques basically turn any SystemC platform model into an LT platform model.

These techniques can be enabled or disabled from the SCML modeling objects which in turn can be used to configure the model in what is referred to as `full simulation` mode and `speed optimized` mode for an loosely-timed platform. Basically, these indicate whether an initiator will initiate a TLM communication call for each data exchange or whether it will use the DMI interface and temporal decoupling to reach an optimal simulation speed.

TLM communication interfaces address the performance overhead of exchanging data between components and temporal decoupling that is usually associated with loosely-timed modeling. The modeling style also addresses the AT modeling style requirements where there is a need for performance improvements due to the much finer grain synchronization. To enable the use of SystemC-based platforms for the performance optimization and architecture exploration, use cases the SCML modeling style extends two more optimization techniques available in SystemC and TLM2.0. These are:

- Clock interfaces:

  Traditional SystemC models will use a `sc_clock` to model clocked behavior. This is a standard SystemC object built on top of the event and process scheduler in SystemC. These clock objects deliver extremely poor simulation performance. The Synopsys SystemC simulation engine provides an improved implementation for those. However, this can be improved further through the introduction of a clock interface that provides more information about the clock and that allows to schedule behavior in the future. The benefit of this is that it reduces the number of processes and events in the simulation by providing a central clock scheduler for all components that rely on the same clock rather than reusing the basic SystemC kernel for that.

- Timing annotation:

  Basically, this technique is the same as temporal decoupling. In the TLM2.0 standard, temporal decoupling is typically linked to the loosely-timed coding style, in the modeling style, timing annotation is used in combination with a quantumkeeper, as provided for temporal decoupling. While the granularity of temporal decoupling is much smaller there is still a performance gain to be had.

Based on the basic simulation technology described in "Optimizing Simulation Performance for FT Models" on page 161, the following rules should be considered while creating SCML component models moving from how to use the SystemC kernel up to the optimization process:

- Kernel activity:

  Minimize the interaction with the process and event scheduler; this should be the first concern of any model architecture since this is the key cause of simulation performance degradation. As already explained, there have been several enhancements made to SystemC specifically with this goal in mind. However, this does not imply that minimizing kernel overhead should be the number one concern with regards to the simulation performance. Consider the following points:

    - It is not necessary to mimic the internal parallelism and concurrency of a component. It is sufficient to make sure that the interaction with the external world happens at the correct time points. Parallelism and concurrency implies events and processes in SystemC, these are only useful when there is contention; that is, several activities fighting over a resource. There should be a synchronization point modeled only when there is a possible contention (that is, the resource is free and there is activity) when the resource assignment schedule can be determined for a series of activities, based on current inputs. Then, this can be used to avoid synchronization in the SystemC kernel.
    - Use TLM also for component interfaces that are not memory mapped. Use TLM1.0 API with a dedicated payload or use the TLM2.0 base protocol restricted to the data pointer. This will avoid signal events and method sensitivity.
    - When reusing code that intensively uses signal interfaces and `sc_clocks`, it may be worth wrapping the code with a TLM interface and a clock gate, so that the component is only active when there is data exchange.
    - Use SystemC methods which are cheaper than threads; immediate notification is cheaper than event notification or timed notification.

- Clock scheduling:

  A special case of kernel activity is incurred through clock scheduling. Traditional SystemC clocks will cause event and process overhead for every tick of the clock. In many cases, users overcome this by not modeling the clock connectivity and revert to use a model parameter indicating the clock period for that model and use that to schedule events which then mimic the clocked activity in the model. This approach causes a mess when trying to mimic the clock tree of a design through parameter settings. It makes it a nightmare when clocks are becoming programmable and even more when optimizations for power (and hence clock frequency scaling) become a key use case for virtual prototypes. Therefore, the SCML2 modeling objects library provides specialized clock interfaces to propagate the clock period to every component that needs it. Further optimizations are possible when each component registers their clocked behavior with a central clock scheduler, thus avoiding event and process overhead by the SystemC kernel for each of the components in the design. Furthermore, this modeling approach also allows the central clock scheduler to keep track of any period changes for the clock during temporal decoupling and to reschedule the callback requests accordingly. Obviously, this last feature has a performance overhead since additional checks and calculations are required for each callback that is scheduled. The key guideline for clocks is to use the SCML2 modeling objects and then to apply the previous rule to minimize the clock based activity.

- TLM calls:

  TLM calls model the communication between components. Each TLM call needs to be forwarded over the memory and interconnect infrastructure even if these components do not modify or use any of the information provided with the call. This is one of the reasons to come up with the TLM2.0 Direct Memory Interface. While using TLM calls it is important to minimize the number of transport calls to implement the communication. The following are the different ways to achieve this:

    - The most efficient TLM interface is the blocking transport interface since it has only two timing points for a data transfer of arbitrary length and it is implemented with a single interface method call. Therefore, it is the preferred API for the loosely-timed modeling style where speed is most important. However, it can also be used for the approximately-timed coding style. The possible drawback is that it must be called from a `sc_thread`, which is more expensive than methods. And more importantly, when a transaction needs to be converted from a non-blocking transport interface to the blocking transport interface, a `sc_thread` needs to be added in the conversion which adds additional synchronization overhead. Conversion from the blocking interface to the non-blocking interface is cheaper since there is no additional thread and synchronization required than would typically be incurred when using a non-blocking interface to start with. A component can limit itself to only support the blocking interface in the following cases:

    - An initiator that only needs the two timing points supported by the blocking API, or it can use the API for the transactions for which this timing information is sufficient. Typically, the speed improvement will be very limited since most likely the blocking interface will need to be converted to a non-blocking interface somewhere in the interconnect path.
    - A target for which it is sufficient to model the latency behavior. Also, here the speed improvement is limited due to the conversion overhead that is likely to be there.

    In both cases however, the simplified coding style is a reason to expect improved performance (less code, so simpler to maintain and optimize).

  - When using the non-blocking interface, there are typically a number of calls required to complete a transaction. Usually, there is a complete set of calls defined that allow full timing accuracy for all possible protocol state transitions and delays of a certain protocol, for example, they enable to exchange data on a beat per beat for burst transactions. The key speed improvement here is to do as few as possible TLM interface calls. The definitions of the protocol state machines is such that they provide with shortcuts to skip states if they happen to be on the same timing point or when they do not provide any meaningful accuracy improvement for initiator or target. These shortcuts should be used whenever possible.
  - One additional way of reducing the number of protocol states required to execute a transaction is to use the TLM2.0 base protocol. This protocol has four timing points for the non-blocking interface. When this provides all features to model the timing of a component, then the TLM2.0 base protocol should be used. When used in combination with other components that use a more detailed protocol engine, protocol conversion will be inserted. This reduces the potential for speed improvement. The protocol state conversion adds the code that would be needed in case all components used the detailed protocol; but there will also be an additional conversion logic to convert the attributes. When used in combination with the SCML storage objects that conversion is anyway needed.

- Use DMI:

  DMI is a key TLM2.0 speed improvement feature. It should be used as much as possible and should be made switchable so that less and more accurate modes are enabled. A DMI handler can be protocol specific or even component specific to make sure that there is a maximum number of DMI calls done. DMI is typically limited to the loosely-timed coding style but should be considered for the AT case as well. Whenever a blocking call makes sense it is also useful to consider whether the full interconnect path can be skipped via a DMI access. DMI allows for a latency parameter which can be used to identify start and end of the transaction. Still there might be an accuracy impact when doing this which should be considered in the evaluation: the impact of resource contention in the interconnect is not taken into account.

- Use temporal decoupling:

  This is a specific variant of minimizing kernel activity by using timing annotation in the TLM interface calls to model timing rather than to use process and event synchronization for every time advance. Temporal decoupling is the key to the performance of loosely-timed simulations. In the modeling style, timing annotation or temporal decoupling should be used whenever possible.

- Configurability:

  Have an LT/AT abstraction switch. Models should be configurable to run at different abstraction levels. The timing accuracy of a simulation relates to the internal accuracy of the model as well as the timing accuracy of the communication interface it is using. For ease of use and overall consistency, each model should be aware of its own accuracy. This means that it should determine what level of timing accuracy it will enable and in which way it wants to initiate transactions or wants to respond to them. The alternative where a model uses the abstraction of the incoming transaction request to determine the timing accuracy it will provide is acceptable, but tends to be lean to less accuracy than you would expect. Transactions that are handled with less accuracy do not only impact their own timing accuracy but also the accuracy of the following transactions.

  More configurability can be added for complex models, so that the simulation performance can be optimized for the specific use case, although typically an on/off switch for accuracy should be sufficient. A model should only provide support to switch on accuracy once, the complexity to support a switch from accurate to less accurate is too high compared to the very limited use cases where it can be used. This complexity is caused by the handling of timed transactions that are in flight at the moment the accuracy is turned off, additional modeling is required to make sure an inaccurate transaction overtakes a more accurate one.

- Use profiling tools:

  Despite the advises discussed above, the best approach to optimize performance is to profile the models, preferably exercised in a realistic context and focus on the specific performance bottlenecks of the model itself.

  - Use the SystemC debugging tools to the following:
    - Check the number of activations of threads or the number of event notifications.
    - Validate whether the DMI configuration is done correctly.
    - Check the number of TLM interface calls that are done.
  - Use generic C++ profiling tools to check the computational overhead for the model.
    - Use this to validate the results of the SystemC debugging tools in context of the overall performance.
    - Look for computational bottlenecks.

Index